Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem

Matthias Gallé

N° d'ordre : **ANNÉE 2011**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Matthias Gallé

préparée au équipe Symbiose
du laboratoire INRIA Rennes - Bretagne Atlantique

Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem

Thèse soutenue à Rennes
le 15 février 2011

devant le jury composé de :

Alberto APOSTOLICO
Professeur Georgia Tech, Atlanta, USA

François COSTE
Charge de Recherche INRIA, France

Alexander CLARK
Professeur, University of London, London, UK

Jacques NICOLAS
Directeur de Recherche INRIA, France

Eric RIVALS
Directeur de Recherche CNRS, LIRMM, France

Gabriel INFANTE-LOPEZ
Professeur, Universidad de Córdoba, Argentina

Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
– Charles L. Dogson

Colorless green ideas sleep furiously
– Noam Chomsky

```
ATGGCCCGGACGAAGCAGACAGCTCGCAAGTCTACCGGC
GGCAAGGCACCGCGGAAGCAGCTGGCCACCAAGGCAGCG
CGCAAAAGCGCTCCAGCGACTGGCGGTGTGAAGAAGCCC
CACCGCTACAGGCCAGGCACCGTGGCCTTGCGTGAGATC
CGCCGTTATCAGAAGTCGACTGAGCTGCTCATCCGCAAA
CTGCCATTTCAGCGCCTGGTGCGAGAAATCGCGCAGGAT
TTCAAAACCGACCTTCGTTTCCAGAGCTCGGCGGTGATG
GCGCTGCAAGAGGCGTGCGAGGCCTATCTGGTGGGTCTC
TTTGAAGACACCAACCTCTGTGCTATTCACGCCAAGCGT
GTCACTATTATGCCTAAGGACATCCAGCTTGCGCGTCGT
ATCCGTGGCGAGCGAGCATAATCCCCTGCTCTATCTTGG
GTTTCTTAATTGCTTCCAAGCTTCCAAAGGCTCTTTTCA
                              GAGCCACTTA
```
– You (HIST1H3J, chromosome 6)

Acknowledgements

It is standard practice to start acknowledging the advisor(s) and I believe this is a good tradition. Each of my two advisors guided and shaped me and this thesis in a different and complementary way and it would just be wrong not to acknowledge this from the beginning. François let me enough place to diverge from the initial statement of this thesis, into directions that fascinated me the most. With a constant optimism (which was much needed at times) he gave good remarks about science and surroundings and always made enough time for me to discuss any idea thoroughly. Gabriel was always an *ametralladora* de ideas and the interactions over the distance and during my visits in Argentina let me with a huge stack of possibilities to explore.

Alberto Apostolico and Alexander Clark acted as *rapporteurs* and read this whole thing very carefully. Their insightful remarks were helpful and very much appreciated. I would also thank Eric Rivals for honouring me with his role as president of the doctoral committee.

The IRISA/INRIA centre in general and the Symbiose group provided an ideal place to perform research. Dozens of persons made of my stay there a rich, pleasant and funny one. There are too many to enumerate them all, but I would at least write down (in what I think is order of appearance) the names of German Capdehourat, Didier Devarus, Carito Vargas, Robert Guziolowski, Alfredo Illanes, Noël Malod-Dognin, Sidney Rosario, Sagar Sen, Mario Rivero, Guillaume Rizk, Katarzyna Kucharczyk, Joaquin Zepeda, Freddy Munoz, Guillaume Collet, Andrés Burgos, Michael Döhler. Thanks for all the coffe breaks, lunch, brainstormings, campings and company. The Symbiose team in particular would not be what it is without the leadership of Jacques Nicolas. Thank you Jacques for all the good advices, for your participation on the committee, for your accurate words and ideas and of course the 21:30 chocolates. A great thanks to Guillaume Rizk, who did all the running to the École Doctorale and the University library and filling out of paper and that I should have done. Fabio Cunial, Tania Roblot and Didier Devarus gave valuable feedback and improved with their opinions the final version of this manuscript. I must also thank the number of others who have kindly read some of these pages and commented upon them. Rayan Chikhi stayed 90 minutes behind a camera and ensured a great video tape of the defence.

From my first days at Rennes, Pierre Peterlongo was much more than the funny post-doc who helped out wherever I needed. I am happy to have co-authored my first paper with him. He and his family hosted us again during the final days of me as a grad-student, and it was great to be able to add an almost insignificant contribution to the renovations of his house.

At the ground level of the INRIA/IRISA, easy to miss if he does not look for it, the attentive visitor might find the Documentation Centre (IST). Pascale, Anne and Agnès tracked down each and every one of the old or obscure paper I asked them for and did not hesitate do get out an old issue of some journal or to ask in some far away library for some book that caught my attention.

Tania Roblot and Matthieu Perrin hadn't much choice but to interact with me during their internship. I really enjoyed their collaboration and the work together.

A project co-funded by MINCyT, INRIA and CNRS permitted me to visit the PLN group at FaMAF. The work outperformed together with Rafael Carrascosa was key to some of the major ideas presented in these pages. I also would like to thank Matias "Chun" Lee, Franco Luque and Martin(cito) Dominguez for their help and their *mates*. Franco was also my principal port whenever NLP-doubts threatened this ship.

Alberto Apostolico kindly permitted me to attend a Dagstuhl Seminar were I saw a bigger picture and were I enjoyed fruitful discussions with him and Michael Clausen. At the final stage of this thesis, Michel Termier (a few weeks before his retirement), Alain Denise and Yann Ponty welcomed me at Paris to discuss about a possible context-free grammar for DNA. Finally, at the end of 2010 Gonzalo Navarro invited me to stay some days in Santiago (Chile) and I was amazed in discovering a bit more about the work done by his team.

My ex-professors and co-students at FaMAF were present during almost all stages of this thesis. Pedro D'Argenio and Nicolas Wolovick gave me valuable advice when I was about to decide the final place to do my PhD. Daniel Fridlender always answered clearly to any question I had about computational complexity. Each of my visits to Córdoba was pleasant, not least due to the good moments spent with Carlos de la Torre and Walter Alini. Carolina Dania, Javier Valdazo and Alejandro Sanchez came all the way over from Spain to see me wear a suit, and I really appreciated it.

My parents expressed their support and love in lots of different and generous ways. In particular, they crossed an ocean to see me presenting my thesis during a cold european winter. Andrei taught me basic arithmetic operations when we were kids and seeking excuses not to sleep. Without them I certainly wouldn't have done this thesis. Though probably visible only to me, Cecilia appears behind every word of this document. . Thank you Samuel for receiving me every night at the door at my arrival and sharing your Legos with me.

Soli Deo Gloria

Matthias Gallé
Saint-Martin d'Hères – France
May 2011

Contents

Abstract

Motivated by the goal of discovering hierarchical structures inside DNA sequences, we address the Smallest Grammar Problem, the problem of finding a smallest context-free grammar that generates exactly one sequence. This NP-Hard problem has been widely studied for applications like Data Compression, Structure Discovery and Algorithmic Information Theory.

From the theoretical point of view, our contributions to this problem is a new formalisation of the Smallest Grammar Problem based on two complementary optimisation problems: the choice of constituents of the final grammar and the choice of how to parse the sequence with these constituents. We give a polynomial time solution for this last problem, which me named the "Minimal Grammar Parsing" problem. This decomposition allows us to define a new complete and correct search space for the Smallest Grammar Problem. Based on this search space, we propose new algorithms able to return grammars 10% smaller than the state of the art on complete genomes.

Regarding efficiency, we study different equivalence classes of repeats and introduce an efficient in-place schema to update the suffix array data structure used to compute these words.

We conclude this thesis analysing the applications. For Structure Discovery, we consider the impact of the non-uniqueness of smallest grammars. We prove that the number of smallest grammars can be exponential in the size of the sequence and then analyse the stability of the discovered structures between minimal grammars for real-life examples. With respect to Data Compression, we extend our algorithms to use rigid patterns as words and achieve compression rate up to 25% better compared to the previous best DNA grammar-based coder.

Résumé

Motivé par la découverte automatique de la structure hiérarchique de séquences d'ADN, nous nous intéressons au problème classique de la recherche de la plus petite grammaire algébrique générant exactement une séquence donnée. Ce problème NP-dur a été largement étudié pour des applications comme la compression de données, la découverte de structure et la théorie algorithmique de l'information.

Nous proposons de décomposer ce problème en deux problèmes d'optimisation complémentaires. Le premier consiste à choisir les chaînes de la séquence qui seront les constituants de la grammaire finale alors que le second, que nous appelons "analyse grammaticale minimale", consiste à trouver une grammaire de taille minimale permettant l'analyse syntaxique de ces constituants. Nous donnons une solution polynomiale au problème d' "analyse grammaticale minimale" et montrons que cette décomposition permet de définir un espace de recherche complet pour le problème de la plus petite grammaire algébrique.

Nous nous intéressons aux algorithmes praticables permettant de retourner une approximation du problème en un temps suffisamment raisonnable pour être appliqués à de grandes séquences telles que les séquences génomiques. Nous analysons l'impact de l'utilisation de classes différentes de maximalité de répétitions pour le choix des constituants et le compromis entre l'efficacité et la taille de la grammaire finale. Nous présentons des avancées algorithmiques pour une meilleure efficacité des algorithmes hors-ligne existants, dont notamment la mise à jour incrémentale de tableaux de suffixes en cours de recodage. Enfin, la nouvelle décomposition du problème nous permet de proposer de nouveaux algorithmes génériques permettant de trouver des grammaires 10% plus petites que l'état de l'art.

Enfin, nous nous intéressons à l'impact de ces idées sur les applications. En ce qui concerne la découverte de structures, nous étudions le nombre de grammaires minimales et montrons que ce nombre peut être exponentiel dans le pire cas. Nos expérimentations sur des jeux de séquences permettent cependant de montrer une certaine stabilité de structure au sein des grammaires minimales obtenues à partir d'un ensemble de constituants. En ce qui concerne la compression des données, nous contribuons dans chacune des trois étapes de la compression à base de grammaires. Nous définissons alors un nouvel algorithme qui optimise la taille de la chaine de bits finale au lieu de la taille de la grammaire. En l'appliquant sur les séquences d'ADN, nos expérimentations montrent que cet algorithme surpasse tout autre compresseur spécifique d'ADN à base de grammaire. Nous améliorons ce résultat en utilisant des répétitions inexactes et arrivons à améliorer les taux de compression de 25% par rapport aux meilleurs compresseurs d'ADN à base de grammaire. Outre l'obtention de taux de compression plus performants, cette approche permet également envisager des généralisations intéressantes de ces grammaires.

Chapter 1

Introduction

The exponential growth of available DNA sequences in recent years is having a fruitful clash with the deep questions underlying information science. Brooks identified in 2003 recent development in biology as the necessary pressure to finally develop a long-time needed quantification of structural information [39].

This thesis is motivated by automatically learning structural models of DNA sequences. As models, we chose formal grammars. They have since long been used to model the underlying structure of natural language [59] and genetic [219] sequences. Their easy interpretation and rigourous definition make them an appealing formalism. Moreover, in the Symbiose team, good results have been obtained on modelling families of proteins with non-deterministic finite automata [125]. Regular grammars however are of limited expression power and notably fail at capturing long-range dependencies. Our goal was to improve expression power, climbing to context-freeness, this time to structure DNA sequences. Confronted with the definition of what to consider a good grammar, in this first phase we followed William de Ockam's advice to seek for the simplest model. This permits us to avoid to introduce any other learning-bias or domain-knowledge and therefore to keep our approach as general as possible.

Instead of considering the generative power of context-free grammars, we focus on the structure they provide over a single sequence. These choices have led us to the formal problem of *finding the smallest context-free grammar that generates exactly one given sequence*. This decade-old problem [211, 225] has been theoretically studied [51] and has applications in several communities. Traditionally, these applications have been in the fields of data compression and approximation to Kolmogorov complexity. Much more scarcely studied, though promising is the use of this problem in structure discovery and grammatical inference. In recent years, there has been also a trend to use them as a backbone for compressed self-indexes. This thesis is the result of our work of using this *Smallest Grammar Problem* to find small, hierarchical structures over DNA sequences.

In the remainder of this introduction we expose the rationale behind, and the context of this thesis. First, we introduce the use of the linguistic metaphor for genetic sequences and review linguistic approaches in biology, focusing on the use of formal grammars. We then review ways of inferring these models. Finally we shortly state the intuition and formalisation behind the well known Occam's Razor, which lead us to the Smallest Grammar Problem.

1.1 Linguistics of DNA

1.1.1 The Linguistic Metaphor

The metaphor of language applied to genetic sequences is old, diffused and recurrent. It is particularly persistent in the media. One of the first books on molecular biology for a broad audience, in 1966, was titled "The Language of Life"[1]. Former US-president William Clinton claimed at the announcement of the completion of the draft sequence of the human genome, on June, 26th 2000, "Today we are learning the language in which God created life"[2]. Linguistic terms refering to genetic sequences can be found by searching for expressions like "book of life" or "code of life". They are used for arguments ranging from the existence of God to tools to identify extraterrestial non-standard form of lifes [240]. A Los Angeles Time article from 1993 [212] reports:

> "DNA-as-language is one of modern science's most powerful metaphors [...] But maybe language is more than just a metaphor. Maybe – just like Sanskrit, Chinese and English – DNA really is a language, with a grammar and syntax that determines how meaning is created."

and a New York Times article said in 1991 [10]:

> "The scientists are approaching genetic sequences as though they were lengthy passages written in an archaic and largely unfamiliar tongue, borrowing methods from the linguist's tool kit to find a bit of order amid apparent biochemical babble.
>
> [...]
>
> The new theories are a subset of a larger science, computational molecular biology, which is fast becoming one of the trendiest disciplines in biological research. Scientists said they were starting to amass so much information about genes and genetic sequences that it was only through the use of a framework like linguistics that they could interpret the incoming rush of data."

The DNA linguistics metaphor is as old as the discovery of DNA itself. Actually, maybe even older. Horace Judson reports [121] a letter from Friedrich Miescher, a contemporary of Gregor Mendel and the discoverer of nucleic acids — "nuclein" as he named it — although their hereditary functions remained unknown to him. In this letter in 1892 he expressed his intuition that some large molecules that are composed of repeated similar (but different) chemical atoms could be able to transmit the hereditary message, "just as the words and concepts of all languages can find expression in twenty-four to thirty letters of the alphabet".

Similar analogies have captured not only the imagination of the general public, but also that of scientists. This may be partly due to a historical coincidence: the same period that saw the discovery of the DNA double-helix structure and

[1] George and Muriel Beadle. "The language of life: an introduction to the science of genetics" Doubleday Publishing Group

[2] see http://clinton3.nara.gov/WH/New/html/genome-20000626.html for the complete speech

advances in the understanding of how hereditary information is transmitted, also witnessed the development of modern linguistics. But considering genetic sequences as a message may be just natural. The interpretation of DNA as a universal language that is read and interpreted by living beings produced terms such as "transcription" and "translation" to refer to the two main mechanisms of transfer of sequential information in the Central Dogma [74]. Apparently it was one of the discoverers of the transcription process — French biologist François Jacob — who coined the expression "the linguistic model in biology"[3]. However, the question remains as to whether referring to DNA or proteins with linguistic expressions is only an image. Before applying techniques used in natural languages we have to know whether genetic sequences can be suitably modelled by linguistic formalisms or whether we should only "understand this continued appeal to DNA as language or text as a simple simplification — as an attempt to convey a complex process in familiar terms" [198].

1.1.2 Identifying Words

If DNA is not just any language, but a language similar to natural languages, it has to share characteristic features with natural languages. One of the most fundamental concepts is the definition of a *word*. In beginning of 1980, E. N. Trifonov took interest in identifying which were the words for DNA. He characterised recurring substrings and linked them to their biological function. Browsing through publications (more than 400), he and V. Brendel compiled 800 of such oligonucleotides, and published them — together with several indexes, descriptions and tables — in a "gnomic" dictionary [231]. Inspired by Kolmogorov Complexity, Trifonov and his co-authors define a measure of complexity of a sequence (called linguistic complexity) and use this to compare protein sequences to natural language text, concluding that protein sequences are more complex [189]. He reviews [230] techniques for identifying such words, using for example the frequency of their substrings and pays special attention to the possibility that the same DNA sequence can be interpreted in several ways. Phrases such as "togethernowhere" that can be decomposed into four different semantically correct phrases[4] seems to be much more common in the genetic language than in natural language. Trifonov remarks that this multiplicity and overlapping of genetic codes has to be taken into account. Because of this multiplicity of meanings, Argos characterises the language of protein folding as "many forked tongues" [21] and Gribskov warns not to carry the language metaphor too far [104], noticing some characteristics that make genetic sequences different from natural language one. He particularly refers to long-range interactions in the secondary structure of proteins, to evolutionary constraints that should be three-dimensional in proteins rather than linear, and to the misleading concept of consensus as the optimal sequence that carries a message, to name some examples.

Some years later, a small controversy rose when Mantegna and co-workers showed that two typical characteristics of human language were present in non-coding DNA. These were Shannon's information measure and Zipf's law [155]. It was the latter that was mainly analysed and that generated most criticisms.

[3]"The linguistic model in biology". In Roman Jakobson: Echoes of His Scholarship, eds. D. Armstrong and C.H. Van Schooneveld, pp. 185-192

[4]"together nowhere", "together now here", "to get her nowhere" and "to get her now here"

George Zipf formulated his famous law in 1949, relating the frequency of a word to its rank in the frequency table of all words. It is a law in the empirical sense, and it is considered more as an observation that holds for a vast variety of data. Mantegna et al. assume that in the coding part of a DNA sequence, a 3-gram corresponds to a word (which is called a codon, and codes for one amino-acid). They analyse therefore different n-grams (for $3 \leq n \leq 5$) for the non-coding part. Plotting their relative frequency against rank in a double logarithmic scale they deduce that they satisfie Zipf's law, concluding with the "possible existence of one (or more than one) structured biological language — present in non-coding DNA sequences". Their article was advertised by a letter in the Science Magazine [91] and heavily attacked in the same section [34] and elsewhere [33, 52, 117, 232]. Tsonis et al. [232] is particularly determined, concluding that "The inescapable conclusion is clear: DNA sequences show no linguistic properties". The original Mantegna article is however still widely cited (positively). Similar experiments with the same conclusion, but targeting coding regions, were performed by another set of authors at the same time [226] and more recently [239] using repeats instead of n-grams (for uses of Zipf's law in other applications of molecular biology see Searls [219]). Most of the time a word in DNA is interpreted as a *codon*, at least in the coding sequences. Wang et al. [239] propose an original definition of a word as a *maximal repeat* (a substring that does appear at least twice in a different context, see also Sect. 3.2) and analyse words frequency in a wide range of coding and non-coding DNA sequences.

Trifonov's emphasis that a DNA sequence can contain different and overlapping meanings was used in another attempt to link linguistics and DNA. Ji [118] goes as far as postulating a biological isomorphism between biological language and natural language in terms of alphabet, lexicon, sentence, grammar, phonetic and semantic. He also assigns the non-coding part of DNA a semantic function, as opposed to the lexical function of the coding part. He uses the term "cell languages" and links this with the interpretation of the cell as a computer and the possibility of DNA computing. The second of his five "(putative) laws of molecular semiotics" states [119]: "Cell language is isomorphic with human language".

1.1.3 Modeling with Grammars

There seems to be little discussion about the fact of interpreting DNA as a formal language in the most general interpretation. It clearly transmits a message and is generated by a yet unknown machinery. Until now we have analysed two approaches of using the term "linguistics of genetic sequences". The first group uses this term more as a metaphor to obtain inspiration, or as an analogy to describe their discoveries. The second one goes a step further and searches for similarities between DNA sequences and human languages. We have seen that such an approach was not always exempt of controversy.

A third approach consists in analysing the expression capacity of genetic sequences through the lenses of formal grammars and connecting it to available tools and understanding. Such an approach is particularly advocated by David Searls. He has published some excellent overviews in 1997 [217] , 2002 [219] and 2006 [57] (this last one with Chiang and Joshi). We refer to these reviews for references to applications of linguistic-inspired tools to a wide range of biological problems. Conceptually, he argues that the main levels in which linguistics work have their counterpart in molecular biology. This is illustrated by the hierarchy used in language, consisting of the lexical, syntactic, semantic and pragmatic levels. This hierarchy can be mapped respectively to the sequence, structure, function and role purpose of macromolecules (see Searls [218] for further details). Without making profound philosophical claims about the interpretation of DNA as a language, he also shows biological examples of RNA sequences that demonstrate the context-sensitive characteristic of interleaved dependency. His formalisation of *String Variable Grammars* [215, 216] – inspired from indexed grammars and designed to model DNA – is of practical interest. A recent implementation uses modern algorithmic tools to provide an efficient implementation of a subset of the functionalities of these grammars [182]. Computational linguistics are concerned with formalising representations that are learnable (efficiently) [65]. For the same reasons, Searls "seeks formalisms that are just sufficiently elaborate and powerful to encompass the range of phenomena under study, but not more so" [57].

Showing a genetic structure that cannot be captured by a context-free language [70], Collado-Vides gives in [71] a transformational grammar that generates regulatory regions of *E. Coli* and *S. typhimurium* and analyses the predictions that this grammar reveals. The approach of defining a grammar that is biologically meaningful and which permits efficient parsing is also taken by Leung et al. [144] in their definition of *Basic Gene Grammars* and applied to model the promotors of *Escherichia Coli*.

Work about "language of proteins" is much more scarce. However, a review from 2006 [99] uses the term "protein linguistics", reviews historical attempts and analyses some of the difficulties and the importance of such an approach.

Different linguistics tools that can be directly applied on protein sequences can be found on the website of the Center for Biological Language Modeling[5].

Loose et al. [151] use regular grammars to describe a language for Antimicrobial peptides (AmPs), small proteins used by the immune system of eukaryotes against bacterial infections. From a database of previously identified AmPs, they automatically generated about 700 regular grammars describing them. They also generated new, unnatural AmPs which were then designed and successfully inhibited the growth of a bacteria. A similar strategy was followed much earlier in 1984 [37] to model a RNA phage group, but the automata were obtained manually.

A careful overview of different uses of formal grammars in molecular biology, with explanations and pointers to other references can be found in Simon [223, Chapter 4–6].

We point out that *linguistics of DNA* should not be confused with *biolinguistics*, the study of the evolution of languages (see Searls [220] or the Journal Biolinguistics[6]).

We have seen that the use of linguistics on genetic sequences goes well beyond that of a simple metaphor. Using a mathematical model permits to uncover meaningful features of these sequences. The pertinence of formal grammars has been studied and they have been applied with success.

[5] http://www.cs.cmu.edu/~blmt/
[6] http://www.biolinguistics.eu

The question remains, however, as to how to find a correct grammar given only sequences generated by this correct grammar. This is exactly the problem that the grammatical inference community studies.

1.2 Grammatical Inference

The field of grammatical inference treats the problem of learning a grammar that generates a given language. We refer to the recent book of C. de la Higuera that reviews the area of grammatical inference, its goals, its tools and its algorithms [75] and detail here only the points relevant to this thesis.

The roots of Grammatical Inference can be traced to the work of Noam Chomsky [59, 60]: his treatment of natural language with formal mathematical models opened the door to attempts to infer these models and to approach language acquisition by children from a computational perspective. Several concepts of *learnability* — how to decide whether a target language can be learnt — exist, but here we will mostly use the definition of *identification in the limit* or *Gold-learning* [102]. It is a known result that super-finite classes of languages[7] (which include regular and context-free languages) are not identifiable in the limit if the learner is presented with positive data only [102]. But if negative examples are also available then regular languages can be identified in the limit in polynomial time [103]. Another positive result concerning regular languages was obtained by Angluin [11], who defines a model that allows queries during the learning process and proves that regular languages can be inferred if (deterministic finite) automaton equivalency queries are allowed.

For what concerns us, we are interested in learning the more expressive context-free grammars from positive data only. There exists various algorithms that use heuristics to infer such a grammar. If more informative data is available, there exists one positive result, due to Sakakibara [202]. We will first consider the heuristics, and then focus on Sakakibara's result.

1.2.1 Learning CFG from Positive Data Only

Despite the negative result concerning learnability of context-free grammars from positive data only, a rich range of algorithms have been designed and applied with success in practice on natural language.

Frequency of words is an important variable in learning, and Wolff [243] uses an algorithm that takes into account frequency of digrams (similar to RePair presented in Sect. 2.6.7) to learn syntax and meaning over data. Another widely-used idea is Z. Harris' concept of *substitutability*. The intuition behind this concept is that strings that appear in the same context are likely to be substitutable, which is translated for formal grammars by saying that they should be constituents of the same non-terminal. Among the algorithms that implement Harris substitutability as their main feature are ABL from van Zaanen [235], EMILE from from Adriaans et al. [5][8] and ADIOS from Solan et al. [224]. A formalisation of this concept, together with other arguments (such as frequency of words or mutual information of the context) also appears in Clark [64], Clark et al. [66] to learn subclasses of context-free grammars.

[7]These are classes that contain all finite languages and at least one infinite language.

[8]For a comparison between both see van Zaanen and Adriaans [237]

These algorithms have to resolve two complementary problems: which are the words that are going to be the constituents of the final grammar, and how will these constituents be used to parse the sentences. Consider for instance the ABL algorithm. It consists of two phases: Alignment learning and Selection learning. The goal of the first part is to extract possible constituents, called hypotheses. ABL does so by aligning the sequences and clustering the unequal parts of the sequences. The selection learning phase takes all these hypotheses and resolves conflicts between contradictory ones. A contradiction here means constituents that overlap. In the prospective part of his thesis [236], van Zaanen considers the possibility of using the equal parts of the Alignment phase as constituents. A similar division is proposed by EMILE, which consists in a clustering phase finding basic rules, and a induction phase generating rules from them. In ADIOS, the MEX procedure distills statistical significant patterns which are put into equivalence classes in a second generalisation phase. Of course, below this high-level view, all algorithms differ significantly. But as we will see, this division reflects well the separation we propose in this thesis: first choosing the constituents, and then deciding which occurrences of each constituent to replace.

Another approach is proposed by Nevill-Manning: in his thesis [171] he proposes different ways of generalising the output of his SEQUITUR algorithm (see Sect. 2.6.3), an algorithm that generates a context-free grammar whose language is exactly the sequence given as input. In his first generalisation non-terminals are merged according to a MDL principle. There are two kinds of merging. The first type consists in merging non-terminals that appear at the same positions at the right-hand side of a rule. The second kind consists in merging the rule bodies if the left-hand side is identical. He then considers how to include recursion into the SEQUITUR algorithm. The last approach detects symbols of the final grammar that predicts another symbol in the future, with a possible gap between both occurrences. This can be interpreted as detecting repeats (over the final grammar) with possible variable gap lengths. Each of these possible generalisations is targeted to one example of a specific application.

1.2.2 Learning CFG from Structural Descriptions

We mentioned before that positive results concerning the learnability of the whole class of context-free languages are scarce. In his thesis, Rémi Eyraud [85] enumerates seven properties that make context-free grammars difficult to learn in polynomial time. For example, a direct extension to context-free of Angluin's algorithm [11] (that learns regular languages) seems improbable because the equivalence problem for context-free grammars is undecidable (property two). The last of these properties is the "structural property", the fact that the structure given by a context-free parse seems much more complicated than the structure of a regular parse. Y. Sakakibara proved in 1992 a remarkable result, which may imply that this last property captures all the difficulty of the learnability of context-free languages:

Theorem 1 (Sakakibara [202])**.** *The class of context-free languages can be learnt in polynomial time from positive samples of structural descriptions.*

A *structural description* is a unlabelled parse tree of the grammar. A learning algorithm could then be designed that would take as input only positive data, infer a parse tree for each sequence and then apply Sakakibara's learning algorithm. In his thesis Eyraud considered this approach, using the SEQUITUR algorithm (see Sect. 2.6.3) to infer the parse trees. Eyraud concludes his study with a negative note, but it is not clear whether the problem lies in the general approach (as the author supposes), in the use of SEQUITUR (which poses problem because it greedily selects the first appearances from the left), in the (only) example used (learning of the language $\{a^n b^n : n > 0\}$, a classical toy example), or a combination of them (using a left-biased algorithm to learn a centred-bracket grammar).

Our choice of modelling genetic sequences by formal grammars poses the challenge of inferring a correct grammar. We have seen two attempts for the case of context-free grammars, the class we focus on. The first one tackles the general problem and infers a context-free grammar from the given positive data through the formalisation of linguistics concepts such as substitutability. The second approach focuses on finding the correct context-free *structure* for each sequence, and resolves the generalisation step with Sakakibara's algorithm. In both cases, a solution to the subproblem of learning the context-free structure of a single sequence would imply major advances. In the first case, we have reviewed attempts of generalising a priori such a structure. For the second case, similar approaches to the one of Sakakibara could be developed. To be able to apply an algorithm inferring a context-free structure on any kind of sequence, we would like to remain as general as possible. Therefore, we follow the ancient intuition of aiming at conciseness and focus on learning a smallest context-free grammar generating a given sequence.

1.3 Occam's Razor and MDL principle

Learning and compressing are deeply connected: when we learn, we are able to express some (possibly infinite) set of data with an explanation which is (generally) shorter than the enumeration of the items. Conversely, compression techniques try to figure out redundancies in the text and a way of doing so is by finding a small explanation for it. For a more detailed but still easy to read tutorial on the relationship between compression and learning, we refer to [4]. For a criticism of this intuition, see Domingos [77].

This intuition has been used for centuries. Attributed to Franciscan friar William of Ockham[9] (c. 1288 – c. 1348), the **Occam's Razor** states in his most famous version that "entities must not be multiplied beyond necessity" and its use in practice can be translated as "if an event is explained equally well by two theories, the simpler one is likely to be the correct one". Or like the medical adage "when you hear the sound of hoofbeats, think horses, not zebras". In molecular biology, it justifies the use of the minimal edit distance in sequence comparison, and the use of parsimony for the construction of phylogenetic trees.

Occam's Razor is neither a theorem, nor a formally defined concept. It is much more a guide, a rule of thumb, that is used intuitively and underlies several formalisations of learning and inference paradigms. The **Minimum**

[9] Who probably was inspired by Aristoteltes who says in *Posterior Analytics*: "We may assume the *superiority ceteris paribus* [other things remaining equal] of the demonstration which derives from fewer postulates or hypotheses – in short, from fewer premises"

Description Length (see Grünwald [108] for a comprehensive recent overview) advocates an inference process resulting in a model such that both the sum of the *description length* of the model plus the description length of the original data with the model is minimised. It states that the best hypothesis H for some given data D is the one that minimises

$$|encoding(H)| + |encoding(D|H)|$$

the length of the encoding of H plus the length of the encoding of D knowing H. The main feature of MDL is that it permits model selection, without falling in the pitfall of *over-fitting* this model to the available data. In a similar direction, the Occam's Razor Theorem [31] gives a formal proof of learnability[10] of a class if there exists a procedure for inferring a smallest hypothesis for this class.

One of the main advantages of an approach inspired by Occam's Razor is that it does not use any other learning bias than simplicity. This seems particularly useful if no (or few) background knowledge is available over the chosen domain. For example, while in the last decades biology has advanced a lot in its understanding of *coding DNA*, the function and purpose of *non-coding* sequences, initially called *Junk DNA*, is much less understood and new knowledge has to be discovered from scratch [194]. However, in the human genome the non-coding part of the sequence represents as much as 98% of the total DNA of an individual. In 1997, Rivals et al. [196] used Occam's Razor as a justification to use a specifically designed DNA compressor to detect approximate tandem repeats in yeast chromosomes.

To summarise, inferring a context-free structure over DNA sequences poses major challenges, and several options have been analysed in the literature. Like in Sequitur, we use Occam's Razor to focus on a minimal model. We restrict this thesis to the search of a smallest grammar of a single sequence as a first step towards a more general inference algorithm. This general algorithm can be achieved by generalising the final grammar, or by using it as a structural description of the given data. In an attempt to be as generic as possible and to be able to apply it on sequences like the non-coding regions of DNA, we search for a smallest grammar. The main subject of this thesis is therefore formalised in the Smallest Grammar Problem, the problem of finding a grammar of smallest size generating only the given sequence.

1.4 Overview of this Thesis

This thesis presents our work on the Smallest Grammar Problem. We arrived to this problem after formalising our motivation to discover new interesting structures on DNA sequences, especially on the non-coding segments. But the Smallest Grammar Problem is of independent interest and has been studied in different areas. Most of the results we will describe here are general and apply to any kind of sequence, but even so we put special emphasis on possible applications to genetic sequences.

Our main theoretical result is a new formalisation of this problem in form of a complete and correct search space (Theorem 5 on page 75). This search

[10]with a definition of learnability called Probability Approximate Correct

space is based on the decomposition of the problem into two complementary optimisation problems. The first one consists in choosing which substrings of the sequence will become the constituents of the final grammar. The second one is concerned with how to combine these substrings in an optimal way. Thanks to this decomposition, we are able to define new algorithms that outperform the state of the art one by 10% regarding the final grammar size. We also present algorithmic improvements on existing off-line algorithms, which include a careful in-place update of an enhanced suffix array. Finally, we consider different applications to which the Smallest Grammar Problem can be applied. In particular, we analyse the different steps of a grammar-based data compression algorithm and present a DNA compressor that outperforms any other grammar-based compressor. The outline of this thesis is as follows.

In **Chapter 2** we state the Smallest Grammar Problem and review the work done on it. We identify three areas of application (Data Compression, Kolmogorov Complexity and Structure Discovery) and present the different approaches to the Smallest Grammar Problem in each of these areas. A special emphasis is given to the algorithms used to obtain a small grammar representing one sequence, and we compare their performance regarding the final grammar size.

In **Chapter 3** we study the choice of constituents with a special emphasis on the trade-off between the quality of the set of constituents and the total time consumed by the algorithm. First, we consider the impact of reducing the universe of possible constituents using different notions of maximality of repeats. Second, we consider the implications of overlapping occurrences. Combining these improvements allows us to reduce the computational complexity of existent algorithms. Finally, we present a data structure (the *double-linked enhanced suffix array*) which we use to compute the set of constituents at each step of the main algorithm, and which can be updated efficiently after each iteration.

Chapter 4 is concerned with the second sub-problem into which we decomposed the Smallest Grammar Problem. Namely, once the set of constituents is given, how to parse in an optimal way the sequence *and* the constituents. We formally define this *Minimal Grammar Problem* and give a polynomial algorithm to solve it. We then use this algorithm to define new approximation algorithms for the Smallest Grammar Problem that improve over the current state of the art.

In **Chapter 5** we come back to the applications we identified. Regarding Structure Discovery, our new formalisation of the Smallest Grammar Problem allows us to analyse the impact of the non-uniqueness of the smallest grammar in this application. We evaluate our algorithms for approximating Kolmogorov complexity through the use of the Normalised Compression Distance to cluster biological sequences. We put special attention to the third application, Data Compression. Analysing different ideas to use grammars for compressing, we present a DNA-focused compressor that outperforms present grammar-based DNA compressors. The use of a special kind of inexact repeats, called maximal rigid patterns, enables us to improve even more our compression capacity.

In the final **Chapter 6** we summarise our contributions, discuss our approach and analyse future directions.

Appendix A gives an overview of the corpora used to validate and compare the algorithms.

Chapter 2

The Smallest Grammar Problem

Formal grammars originated with the purpose of describing a language, a possible infinite set of strings. At the same time, this description by the grammar acts not only as a generator, but permits also to describe an underlying structure of the language. The Smallest Grammar Problem puts its focus on structuring a single sequence, and consists in finding the smallest context-free grammar that generates exactly this sequence. This chapter is devoted to the analysis and review of approaches tackling this problem. Before starting, we introduce our notations and give some definitions. In Sect. 2.2 we give an overview of the origins of the Smallest Grammar Problem and of the motivations behind the research communities that studied it. The next three sections focus on the work on the Smallest Grammar Problem motivated by applications in Data Compression (Sect. 2.3), Kolmogorov Complexity (Sect. 2.4) and Structure Discovery (Sect. 2.5). In all these sections we will make references to different algorithms, all of which are detailed afterwards, in Sect. 2.6. Finally, in Sect. 2.7 we define a framework that generalises most of these algorithms. This framework enables us to compare the different algorithms in a uniform setting. We perform an exhaustive comparison, evaluating their ability to return small grammars on different types of sequences. In a second comparison we review grammar-based algorithms that have been used for DNA compression and compare their performances.

2.1 Definitions

We introduce the notation and definitions used in this thesis. Most of it is standard, except maybe our notation for (non-overlapping) occurrences (page 12) and the definition of straight-line grammars (Sect. 2.1.3).

2.1.1 Sequences

A *sequence* s is a concatenation of zero or more characters from an alphabet Σ: $s \in \Sigma^*$. $\Sigma(s)$ denotes the alphabet set over which s is drawn. The number of characters in alphabet Σ is denoted by $|\Sigma|$. We will denote single characters or strings by single letters, so concatenation is denoted by simply concatenating symbols. $(w)^k$ denotes the sequence of length $k|w|$ which is w concatenated k times. We will also use $\prod_{i=1}^{n} w_i$ to refer to the sequence $w_1 \dots w_n$. For example, $a^k = \prod_{i=1}^{k} a$.

We start indexing sequences from 0. So, a sequence s of length n over the alphabet Σ is represented by $s[0]s[1] \dots s[n-1]$, where $s[i] \in \Sigma \; \forall 0 \leq i < n$. We denote by $s[i,j]$ $(i \leq j)$ the sequence $s[i]s[i+1] \dots s[j]$ of length $j - i + 1$. If $j < i$ then $s[i,j] = \epsilon$, the **empty string**. Furthermore, the sequence $s[0,j]$ $(0 \leq j < n)$, also denoted by $s[..j]$, is called a **prefix** of s, and symmetrically, $s[i, n-1]$ $(0 \leq i < n)$, also denoted by $s[i..]$, is called a **suffix** of s. We say that sequence $s[i,j]$ **occurs** at position i in s and that it is a **substring** of s. In general we will use letters s, t for general strings and v, w for substrings.

Given a sequence s, $\boldsymbol{pos}_s(w)$ denotes all the positions where string w occurs in s (the **occurrences** of w). Two different occurrences of w (let say, i and j, with $i < j$) **overlap** if $i+|w|-1 \leq j$. An important role in this thesis are played by non-overlapping occurrences of substrings. There are several ways of selecting occurrences such that the selection does not contain overlapping ones. We will call the **normalised non-overlapping occurrence list** (denoted $\mathcal{L}_s(w)$) the list of occurrences defined in a greedy left to right way as follows. First, choose the leftmost occurrence. Next, select the following leftmost occurrence that does not overlap with the previous one. Continuing until the last occurrence, the resulting selection will contain a maximal possible number of non-overlapping occurrences.

A **repeat** of s is a substring of s that occurs more than once. $\mathcal{R}(s)$ denotes the set of all repeats of s, while $\hat{\mathcal{R}}(s)$ reduces this to the set of all non-overlapping repeats of length at least two: $\hat{\mathcal{R}}(s) = \{w : |\mathcal{L}_s(w)| > 1 \wedge |w| > 1\}$.

In some cases it will be useful to specify a *separator* symbols, over which no repeat can span. Therefore, we suppose that the symbol $|$ denotes a new symbol every time it appears. For example, $ab|cb|cd = ab|_1cb|_2cd$.

2.1.2 Grammars

Our exposition here follows loosely the classical work of Hopcroft and Ullman [115].

Formal grammars are rewriting systems that permit to generate a set of strings starting from a single symbol. In their most general form, a grammar G is a 4-tuple $\langle \mathcal{N}, \Sigma, \mathcal{P}, S \rangle$. Σ and $\mathcal{N}$ are disjoint, non-empty sets of symbols called respectively **terminals** and **non-terminals**. To refer to a non-specified member of any of these sets we use the term **symbol**. S is a special non-terminal called the **starting symbol** or **axiom**. $\mathcal{P}$ is a subset of $(\mathcal{V} \cup \Sigma)^+ \times (\mathcal{V} \cup \Sigma)^*$. A member of $\mathcal{P}$ is a **rule** (or **production**) and denoted by $\alpha \rightarrow \beta$. α is the **left-hand side** and β the **right-hand side**. In general we will denote with greek letters strings from $(\Sigma \cup \mathcal{N})^*$, with lower-case latin letters strings from Σ^* and with upper-case latin letters symbols from $\mathcal{N}$.

We say $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$ whenever $\alpha \rightarrow \beta$ is a production and denote by $\stackrel{*}{\Rightarrow}$ the reflexive and transitive closure of relation $\Rightarrow$. The **language of a non-terminal** is defined by the set of terminal strings that can be produced from it: $L(N) = \{w \in \Sigma^* : N \stackrel{*}{\Rightarrow} w\}$ (the *constituents*). The **language of a grammar** is the language of the start symbol: $L(\langle \mathcal{N}, \Sigma, \mathcal{P}, S \rangle) = L(S)$.

Different classes of grammars are defined by restricting the allowed rules. If there is no additional restriction on the set of production rules, the class of grammar is called the class of **unrestricted grammar** which are equivalent to Turing Machines [115]. A **context-sensitive grammar** requires each right-hand side to be at least as long as its left-hand side. Each such grammar has a normal form where each rule is of the form $\gamma N\delta \rightarrow \gamma\alpha\delta$, with N a single non-terminal and $\alpha \neq \epsilon$. γ and δ act as "context" for this production. Some definitions permits a special rule $S \rightarrow \epsilon$ to enable context-sensitive languages[1] to contain the empty word. The main class we will consider here are **context-free grammars** whose production rules have to be of the form $N \rightarrow \alpha$, with N a single non-terminal. A context-free grammar is in **Chomsky Normal Form** (CNF) if every production is of the form $N \rightarrow AB$, $N \rightarrow a$ or $N \rightarrow \epsilon$. Traditionally, the most restrictive class are **regular grammars** with rules of the form $N \rightarrow N'a$ or $N \rightarrow a$, with N, N' non-terminals and a terminal. Regular languages are exactly recognised by the class of finite-state automata.

The language generated by each of this classes is strictly contained in the previous. This hierarchy is called the *Chomsky* (or *Chomsky-Schützenberger*) hierarchy.

2.1.3 Straight-Line Grammars

The Smallest Grammar Problem focuses on grammars that generate exactly one sequence. We define here a class of grammars with this characteristic.

There should be only one production rule per non-terminal[2]. Also, focusing on context-free grammars, this means that no recursion should be possible. If not, this would result in an infinite production as no choice is possible in a context-free grammar with one rule per non-terminal.

We define therefore **straight-line grammars**:

Definition 1 (Straight-Line grammar (SLG))**.** *A straight-line grammar is a grammar such that:*

1. *every non-terminal appears at the left-hand side of at most one production rule*

2. *Given the graph $G = \langle \mathcal{N}, E\rangle$, with $(N, N') \in E$ if N' appears in the right-hand side of the rule of N, G has to be acyclic.*

The term straight-line comes from the fact that the parses of such grammars do neither branch (this would violate Condition 1) nor loop (Condition 2).

A grammar without branches that loops permits only one infinite derivation and has therefore an empty language. This motivate the following alternative characterisation:

Proposition 1. *Suppose a grammar G that satisfies Condition 1 of Def. 1. G is straight-line if and only if $|L(G)| > 0$*

[1] A language is context-sensitive if it can be generated by a context-sensitive grammar.

[2] It is possible to violate this condition and still generate a single sequence. This could be interesting to permit alternative parses of the same substring, but as we are going to focus on the final size of the grammar, these rules could be replaced by a single rule obtaining a smaller grammar.

A straight-line grammar in Chomsky Normal Form is equivalent to a straight-line program. Because in this thesis we are mostly interested in the structure given by the grammar, we will in general not consider our grammars to be in CNF.

Proposition 2. *Let G be a SLG. Then $|L(G)| = 1$ and moreover, $|L(N)| = 1$ for all non-terminal N.*

We will denote by **constituent** of N ($cons(N)$) the only string of $L(N)$.

Except otherwise stated, throughout this thesis, the term grammar will always stand for a context-free and straight-line grammar.

In general, the non-terminals of our grammars are *anonymous*, which means that their only meaning is to differentiate them from other non-terminals. They can then be re-defined if sufficient care is taken to modify equally non-terminals in the same way. In particular, we will suppose that the production rules can be enumerated $N_1 \to \alpha_1, \ldots, N_{|\mathcal{P}|} \to \alpha_{|\mathcal{P}|}$ like follows. $N_1 = S$, N_2 is the first non-terminal that appears at the right hand side of the S rule and in general N_{i+1} is the i-th different non-terminal that appears in $\prod_{j=1}^{i} \alpha_i$. We discard non-terminals that are not used in any production rule (the only exception is for the definition of straight-line grammars with don't cares in Sect. 5.4.2).

If G is a SLG, then $r(G)$ will denote its **canonical sequential representation** as: $\prod_{i=1}^{|\mathcal{P}|}(\alpha_i \$)$, where \$ is a special end-of-rule symbol that does not appear in G. G can be recovered unambiguously from $r(G)$ and it seems to be the most intuitive way of representing a SLG linearly. Therefore we define the size of a grammar to be the size of its canonical sequential representation:

Definition 2 (Size of a SLG)**.** *If G is a SLG, then $|G| = |r(G)|$.*

Therefore, $|G| = \sum_{N \to \alpha \in \mathcal{P}} (|\alpha| + 1) = \sum_{N \to \alpha \in \mathcal{P}} (|\alpha|) + |\mathcal{P}|$

Finally, note that from the set of production rules $\mathcal{P}$ alone, the whole grammar can be recovered: $\mathcal{N}$ is the set of left-hand sides, Σ is composed of the remaining symbols, and S is the non-terminal that derives the longest terminal string. So, we can use as indistinguishable $\mathcal{P} = \mathcal{P}(G)$ and $G = G(\mathcal{P})$.

Now we are able to state our main problem:

Definition 3 (Smallest Grammar)**.** *Given sequence s, a straight-line grammar G^* is a* **smallest grammar** *if $L(G^*) = \{s\}$ and $|G^*| \leq |G|$ for any other straight-line grammar G such that $L(G) = \{s\}$*

The **Smallest Grammar Problem** (SGP) is the problem of finding a smallest grammar for a sequence s.

2.2 Origins of the Smallest Grammar Problem

We could trace two independent origins for the idea of representing only one sequence by a grammar. The first appearance of this concept we could find was in a seminar hold by the Psychological Society of the former GDR in 1973 [211]. The idea to describe objects by a minimal set of rules was used in the 1960s by Emanuel Leeuwenberg to define *Structural Information theory*, a similar theory to *Algorithmic Information theory.* Coming from the cognitive psychology he focuses on how human perception identifies and uses this minimal description of visual objects. The seminar of 1973 contains several contributions that use context-free grammars as models to describe visual objects and that study the relationship of a minimal grammar to human learnability with that object. Such a minimal context-free grammar was then used as a computable approximation of Kolmogorov complexity. Ebeling and Jiménez-Montaño [80] use this in 1980 to define the grammar complexity of a string and to study the inherent complexity of genetic sequences.

The second source originates in the data compression community, inside the bigger schema called *macro* or *dictionary-based*. Storer and Szymanski define in 1982 [225] several such compression techniques, including one that maps to a context-free grammar and prove that the the problem of finding a smallest such grammar generating exactly one sequence is NP-Hard.

Later, Nevill-Manning and Witten [172] introduce their SEQUITUR algorithm and praise its capacity of generating a small context-free grammar that describes well the underlying structure of the given sequence. Shortly after, Kieffer and Yang [127] analyse the compression capacity of what they called *Grammar-Based Codes* from an information theory point of view.

More recently, in 2002, Charikar et al. [50] state again the relationship of a minimal grammar to Kolmogorov Complexity. The thesis of Lehman [141] and the complete paper of Charikar et al. [51] builds upon Storer and Szymanski result and analyses approximations to a smallest grammar. With respect to hardness, two more insights are given: in first place they show that — supposing $P \neq NP$ — there is no polynomial algorithm that can ensure an approximation better than $\frac{8569}{8568}$ in the worst case. Moreover, they unveil a relationship to *addition chains*, a decade-long studied algebraic problem. Any algorithm that would ensure an approximation ratio of $o(\log n / \log \log n)$ would be a progress into the problem of finding the shortest addition chain that contains a given set of integers.

In what follows we present three applications to which the Smallest Grammar Problem has been applied. The first is Data Compression and is based on the insight that it may be cheaper (in terms of number of bits) to send a small straight-line grammar instead of the original sequence. We pay special attention to the use of such grammars inside the general topic of compressed data structures. The second application is the approximation to Kolmogorov Complexity and reflects well the original motivation of the problem. Finally, we consider Structure Discovery. In all cases, we introduce the general research field and show how the Smallest Grammar Problem has been tackled in this field.

2.3 Data Compression

In general terms, Data Compression is concerned with finding an encoding of data that requires less bits than just spelling out the original data. The existence of a *decoder* is essential to recover the original data from the encoded bit string. If the decoded object does not correspond exactly to the original one we talk of *lossy compression*. Lossy compressors are widely used in fields like image and audio treatment, where the final decoded object can be degraded without concern for a human user. Here, we will focus on lossless data compression.

Traditionally, lossless data compression algorithms are divided into two categories: macro-based and statistical. The first group seeks redundancies in the text by detecting repeated patterns, and compresses the sequences by replacing an occurrence of such a pattern with pointers to a previous occurrence. They achieve good compression by replacing subwords with (shorter) references. Statistical-based compression algorithms are based on information theory and assign codewords to single symbols. They are based upon the insight that it is better — for compression purpose — to assign shorter codewords to frequent symbols. This relation between codeword length and frequency is formalised in Shannon's *noiseless coding theorem* that says that, given a source i.i.d with probability p that produces an infinite stream of symbols $\omega_1 \ldots \omega_n$, an optimal code would have codewords $c_1 \ldots c_n$ such that $|c_i| = -\log p(\omega_i)$. We refer to the classical work of McKay [163] for further reference.

2.3.1 Dictionary based

A good overview of different possible frameworks of macro schemes is given in the work of Storer and Szymanski from 1982 [225]. There, the authors differentiate between *external* and *internal* macro schemes. External macro schemes contain pointers to an external dictionary, while the pointers of internal macro schemes point to positions of the sequence itself. Our definition of LZ78 (see Sect. 2.6.2) defines it as an external macro scheme, while LZ77 (Sect. 2.6.1) is internal.

From the external macro schemes, we will pay special attention to a class of compression algorithms called **fixed size dictionary**. In this framework, the dictionary consists in a set of words $\{\omega_1, \ldots \omega_n\}$. Each dictionary word has an associated codeword $C = \{c_1, \ldots, c_n\}$. The set C must be uniquely decodable (prefix-free or fixed-length for example), and it is linked to the dictionary by the function f defined as $f(c_i) = \omega_i$. The goal is to find $d_1 \ldots d_m$ such that $d_i \in C$, $f(d_1) \ldots f(d_m) = s$ and $\sum_{i=1}^{m} |d_i|$ is minimal. This problem was proposed in 1973 by Wagner [238] together with a dynamic algorithm that solves this problem in an optimal way. The problem was called "optimal parsing" by Bell et al. [26] and "Minimal Space" by Schuegraf and Heaps [213] where it is solved by a shortest-path algorithm. There also exists faster approximate algorithms: see Katajainen and Raita [124] and Bell et al. [26, Chapter 8.1]

Interestingly, in his seminal work, R. Wagner interprets the fixed size dictionary in a grammatical way [238]:

> "The set of phrases given initially acts like a partial grammar for a context-free language. The language consists of a finite set of sentences (the phrase and message strings themselves). The right-hand sides of its grammar rules contain no non-terminal symbols."

Storer and Szymanski [225] characterise a richer schema and give NP-hardness proofs for several variants. This include variants where the pointers may be recursive (this is, enabling phrases itself to be parsed with pointer to other phrases) or the phrases may overlap. They show that any of the four alternatives that combines this restrictions results in an optimisation problem which is NP-Hard. This includes the problem of finding a context-free grammar of smallest size.

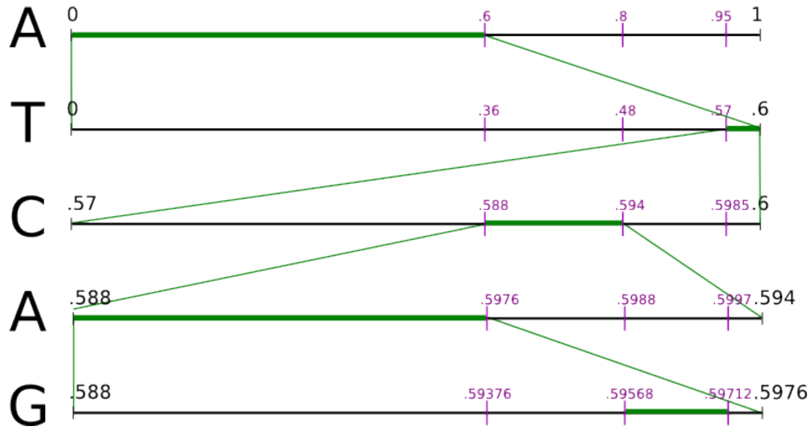


Figure 2.1: Arithmetic coding of sequence $ATCAG$, supposing $p(A) = 0.6, p(C) = 0.2, p(G) = 0.15, p(T) = 0.05$.

2.3.2 Statistical methods

Statistical compression algorithms do not replace subsequences with pointers but encode one symbol at a time. They have the advantage that they permit to divide the coding process into two. On one hand the actual encoding, that takes as input a probability distribution p and encodes the current symbol c with a codeword corresponding to $p(c)$. On the other hand there is the modelling of the sequence and inferring of the distribution p, which does not need to consider the mapping from probability to codewords. This proves to be particularly useful for implementations, where a good encoder needs to be implemented only once, and then different probability models can be tested.

Though Huffman Coding is probably the easiest to understand statistical encoder, we will review here **Arithmetic Coding**, which we will use in this thesis. An arithmetic coder encodes a string with one real number (between 0 and 1). Suppose a probability distribution over the DNA alphabet p such that $p(A) = 0.6, p(C) = 0.2, p(G) = 0.15, p(T) = 0.05$. The interval $[0, 1]$ is then divided accordingly (see Fig 2.1). Any real number between 0 and 0.6 represents the sequence $s = A$, while 0.73 for instance represents $s = C$. Suppose the first symbol of s is A. In this case, the interval $[0, 0.6]$ is again divided according to p. Now, 0.4 stands for $s = AC$ and 0.58 for $s = AT$. See the rest of Fig. 2.1 for a bigger example. The final real number is then encoded in binary. Two issues differentiate arithmetic coding algorithms. In first place in the example we just presented there is no way of knowing how long the sequence is. 0.5 may stand for $s = A$, $s = AG$, $s = AGA$, etc. Two main solutions exist: a special sentinel symbol can be added (with very low probability) that occurs only once, at the end. Or, the real length of the sequence is sent at the beginning. The second issue is the choice of which of the infinite real numbers in the final interval will be used to represent the sequence. There may exist more than one number with minimal binary representation. Finally, for real implementations special care has to be taken with precision and overflow limitations.

An **adaptive arithmetic coder** (AAC) changes the probability distribution while encoding the text. For example, a 0-order arithmetic encoder starts with some fixed distribution ($p(\omega_i) = 1/|\Sigma|$ for example). It keeps a frequency table that counts the number of occurrences of each symbol, and after encoding each symbol, the probability distribution is updated with the empirical distribution of the symbols seen so far. While all the information that is used in this update is contained in the data that was already sent, the decoder can mimic the behaviour of the encoder and decode without error the bit stream. An adaptive encoder has the advantage that no probability distribution has to be sent at the beginning, and that it can adapt to local changes of the input text. We will often use an n-context arithmetic code (**n-AAC**), which models the sequence with a n-gram model.

2.3.3 DNA Compression

Standard compression tools do not compress well DNA sequence. The term "not compress" has to be clarified: as DNA can be considered — without loss of information — as a sequence over an alphabet of four letters, a trivial baseline for encoding such a sequence would take two bits per symbol. If general-purpose compressors are applied to them they result generally in a bitstream longer than $2n$, for a sequence of n nucleotides.

Since this was stated, several different research groups have tried to develop specific algorithms that take into account the peculiarities of DNA sequences. We have counted no less than 20 different algorithms designed for this purpose. Recently, two excellent papers review most of these. Giancarlo et al. [98] make an overview of biological problems where compression techniques have been applied. In a complementary view, glu et al. [101] take the side of the compression community and analyse how concepts developed there have been applied in biology. Arguably, the main motivation of such an effort is not so much the gain in storage space or bandwidth (until the popularisation of High Throughput Sequencing in recent years it was not sure if this was even a problem), but the desire to find some redundancy in the "code of life" that may give insights in the evolutionary pressure or the function of non-coding DNA, to cite just some examples.

The first[3] specific DNA compressor was presented with the algorithm *Biocompress* [106] and shortly after extended with an arithmetic coder of order 2 [107]. Only exact repeats are considered. It consists of a LZ77-style parse over the sequence, where biological palindromes are also considered. A window size is considered for efficiency purposes. Two years later, *Cfact* by Rivals et al. [195] takes an offline approach, consisting of two phases: in the first, a suffix-tree is used to select interesting exact repeats, and in the second the occurrences of selected repeats are evaluated: if their estimation of compression gain is positive, they are replaced by pointers. The authors use this algorithm to detect tandem repeats in the chromosomes of yeast [196]. Another algorithm that considers only exact repeats is presented by Manzini and Rastero [157]. According to the authors its major virtues are its speed and low requirements of space. To achieve this, repeats are identified by using a technique called *fingerprint* of patterns. Special care is used to encode the resulting pointers efficiently. Inexact repeats enter the scene with *GenCompress* [54]. Again, a LZ77-like

[3]Though it does not produce a bit stream which can unambiguously be decompressed, the method for discovering significant DNA by their minimal-length encoding as described in Milosavljević and Jurka [164] dates of the same time.

phase is performed, but supporting edit operations replace, insert and delete. It is applied to reconstruct evolutionary phylogeny tree, as a clear precedent to the definition of the similarity metric (see Sect. 2.4.2). Algorithm *DNACompress* [55], performs similar, but interesting inexact repeats are computed with the aid of the algorithm *Pattern Hunter* [152]. The problem of using non-exact repeats is that the amount of them is much higher then those of exact repeat, and special care has to be taken in selecting which choice of non-overlapping occurrences will be replaced. Chang [49] considers its *DNAC* algorithm to be a mix of *GenCompress* and *Cfact* and compress DNA sequences in four phases: first, it selects a (in our notation, see Sect. 3.2) super-maximal repeats using a suffix tree, then extends it to approximate repeats (edit operations), calculates an optimal combination of non-overlapping occurrences and encodes the final result with Fibonacci code. A similar approach is used in *DNAPack* [25] where dynamic programming techniques are used to ensure that the correct choice of occurrences is made. The good performance of this algorithm can be explained also by the fact that different coding schemas are used and special attention is given to the right choice between them. The Burrows-Wheeler Transform is used by Adjeroh et al. [3] to analyse the nature of the repeats in DNA sequences, interleaving it in different phases of a dictionary-based compression pipeline.

Good statistical encoders arrived later on the scenario of DNA compression, but performed often better. In this setting, the quest is to identify the best probability model for the target sequence. *CDNA* [150] does so by using a variable-length context that takes account of inexact matches, and combines different models. Their final model needs several parameters, which are estimated with Expectation-Maximisation on a pre-established corpus. A simpler schema was used in *ARM* by Allison et al. [8] who re-implement and improve over the model of Milosavljević and Jurka [164]. Some of the authors of *ARM* present *XM* [42], another pure-statistical DNA specific compressor which introduce some new ideas. The probability distribution that predicts the next symbol is given by a combination of different *expert* models. Four different classes of expert models are used: a classical Markov model of order k (the authors use $k = 2$ for DNA and $k = 1$ for protein), a context Markov expert which is a classical context-model restricted to the local history (previous 512 symbols in this case) and two kind of repeat experts, which consider the next symbol to be part of a copied region from a particular offset. The normal copy expert consider standard copies, while the reverse expert consider reverse complement repeats.

A final class are hybrid algorithms, which combine both dictionary and statistical schemes. *CTW+LZ* [161] encodes repeats by one of two methods: with LZ77-like pointers (long repeats) or statistically with a context-tree weighting (CTW) model (shorter repeats). *MNL* [227] and its improved successor *GeMNL* [135] perform a block-parse of the sequence and encode each block with one of three variants: or a direct 2-bit-per-symbol encoding, an order-1 context model or a normalised maximum likelihood (NML) model. The NML model tries to find for each block an appropriate similar block in the already encoded data. Recently, S. Deusdado takes in his thesis [76, in portuguese] the best of previous algorithms and joins them into an algorithm called *DNALight*. Like *DNACompress* it first selects a dictionary of useful repeats (using an algorithm also explained in his thesis), which is itself parsed with the same dictionary. The resulting sequence (the part over which no selected repeat spans, plus the index over the dictionary) is encoded with a statistical encoder using a model similar to *XM*. Global models (of order 10 over codons) are combined with two local models to predict the next symbol.

In Table 2.1 we compare the result of these algorithms on the standard DNA compression corpus. Results are given in bits per original symbol. Absences and less precision than four digits are due to how they were presented in their respective paper (or material available on the web). *DNALight* achieves the best compression on all but two sequences, but as it can be appreciated the difference with others is mostly less than 10^{-2}.

Finally, we should note that there have been a recent trend (even if some work dates already of 2001 [210]) of DNA compression algorithm to emphasise the compression aspect over the learning one. The focus is less on extracting all possible redundancy of the sequences, but to be able to process fast and without much memory requirements, complete databases [137, 241] or genomes [61].

2.3.4 The SGP in Data Compression

One of the striking forces in the development of the work in the Smallest Grammar Problem is their applications to data compression. Straight-line grammars have the attractive characteristic that they provide a neat way of combining the two groups of text compression, dictionary-based and statistical. In a first step, a grammar is inferred from the sequence, based on the repeats inside the sequence. This non-sequential structure can then be transformed into a sequential one, which itself can be compressed with a statistical encoder. The fact that non-terminals are anonymous, that their frequency of appearance vary a lot and that rules can be presented in any order provide opportunities to take advantage of the statistical compressors. Moreover, the hierarchical nature of grammars allows richer models than a mere dictionary of words.

In this sense, a compressor that uses a SLG can be divided into three steps (as we pictured in Fig. 2.2). First, a context-free grammar G_s is generated from the input sequence s. Second, this grammar is transformed into a sequential representation R_s which then is encoded by a compressor into the bit stream B_s which can be transmitted. Note that other applications of SLG — besides compression — generally only consider the first step and the traditional presentation of grammar-based code [127] unifies Step 2 and 3. Of course, as pointed out by Charikar et al. [51], if the size of G_s is n, then the size of B_s can easily be bounded by $n \log n$, assuming a fixed-length code of size $\log n$. A theoretical study that only considers asymptotic behaviour can dismiss hidden constants and this logarithmic factor, but for real data compression algorithms that are suppose to work on finite strings, that may make a big difference.

Some work has been done targeting Step 2. Nevill-Manning et al. [178] introduce a method that sends the right-hand side of a rule the first time a non-terminal is found. The second time it sends a pointer to the first occurrence and from there on it uses a unique identificator. For rules that only appears twice, this method never names the corresponding non-terminal with an absolute identifier, reducing the size of the final alphabet. In the presentation of DNASequitur [56] much care is taken in this second step, and the final result varies accordingly. In the next paragraphs we will see other methods used by Kieffer and Yang that take advantage of properties of a special kind of SLG.

The compression capacity of SLG was studied deeply by a group led by

sequence	BioCompress-2 [107]	GenCompress [54]	CTW-LZ [161]	DNACompress [55]	DNAPack [25]	CDNA [150]	GeNML [135]	XM [42]	DNALight [76]
chmpxx	1.6848	1.6730	1.6690	1.6716	1.6602	–	1.6617	1.6577	**1.6415**
chntxx	1.6172	1.6146	1.6120	1.6127	1.6103	1.65	1.6101	1.6068	**1.5971**
hehcmv	1.8480	1.8470	1.8414	1.8492	1.8346	–	1.8420	1.8426	**1.8317**
humdyst	1.9262	1.9231	1.9175	1.9116	1.9088	1.93	1.9085	1.9031	**1.8905**
humghcs	1.3074	1.0969	1.0972	1.0272	1.0390	**0.95**	1.0089	0.9828	0.9724
humhbb	1.8800	1.8204	1.8082	1.7897	1.7771	1.77	–	1.7513	**1.7416**
humhdab	1.8770	1.8192	1.8218	1.7951	1.7394	1.67	1.7059	1.6671	**1.6571**
humprtb	1.9066	1.8466	1.8433	1.8165	1.7886	1.72	1.7639	1.7361	1.7278
mpomtcg	1.9378	1.9058	1.9000	1.8920	1.8932	1.87	1.8822	1.8768	**1.8646**
mtpacga	1.8752	1.8624	1.8555	1.8556	1.8535	1.85	**1.8440**	1.8447	1.8442
vaccg	1.7614	1.7614	1.7616	1.7580	1.7583	1.81	1.7644	1.7649	**1.7542**

Table 2.1: Comparison of DNA compressors on DNA Corpus. Best for each sequence is boldfaced.

$$s \overset{①}{\Longrightarrow} G_s \overset{②}{\Longrightarrow} R_s \overset{③}{\Longrightarrow} B_s$$

Figure 2.2: Schematic process of a encoder that uses a straight-line grammar: first the grammar G_s is inferred from sequence s. It is then transformed into a sequential representation over the alphabet $\Sigma \cup \mathcal{N}$ (plus eventually some extra symbols) and finally encoded into bitstream B_s

Kieffer and Yang, who named this codes **Grammar-Based Codes** (GBC). They introduce the definition of **irreducible grammars**:

Definition 4 (Irreducible Grammar [127])**.** *A straight-line grammar $G = \langle \mathcal{N}, \Sigma, \mathcal{P}, S\rangle$ is irreducible if:*

1. *G is admissible: it does not generate ϵ and it is pruned from any non-terminal that is not used in its only derivation*
2. *$cons(N) \neq cons(N')$ for all $N, N' \in \mathcal{N}$ and $N \neq N'$*
3. *Each non-terminal appears at least twice: $|pos_{r(G)}(N)| > 1$ for all $N \in \mathcal{N}$*
4. *No substring appears more than twice in non-overlapping positions (excepting single symbols): $|\mathcal{L}_{r(G)}(\alpha)| \leq 1$ for any $\alpha \in (\mathcal{N} \cup \Sigma)$ s.t. $|\alpha| = 2$*

Kieffer, Yang and co-authors define several algorithms that generate Irreducible Grammars (notably, Sequential, Bisection, Multilevel Pattern Matching and a variable of LongestFirst, see Sect. 2.6). But their main focus is on their final compression capacity, and they pay special attention to Step 2 and 3 of Fig. 2.2. For the algorithm that generates the grammar itself (Step 1), they define a set of transformation rule such that successive applications of them transform any grammar into an irreducible one.

Yang and Kieffer [244] define algorithms that read the sequence symbol by symbol, maintaining a grammar that generate the prefix read until there. The grammar is updated after reading each symbol with the set of transformation rule. They then define three different possibilities of encoding this grammar (Step 2 and 3):

1. In the first — called *hierarchical* — Step 2 consists in inserting special symbols b and e at the end and beginning of any rule with right-hand side longer than two. Arranging the rules in the correct order permits an unambiguous decoding of the grammar. As most rules have length two, this produces a smaller representation than $r(G)$. Step 3 consists in a standard 0-order adaptive arithmetic coder.
2. Another encoding is named *sequential*. Here, Step 2 and 3 are interleaved: after reading every symbol not only the grammar is updated, but it is also directly encoded. Simulation results on artificial examples shows that this sequential encoding procedure performs better than the hierarchical procedure.

3. Finally, they use properties of irreducible grammars to improve over this. After each symbol one bit is sent that indicates if the updated grammar is simply the old grammar plus the new symbol concatenated at the end of the axiom rule, or if one of the transformation rules was applied. This bit is then used as context information by the arithmetic coder.

In a second part of this series of papers [245], Yang and Kieffer consider "context models". They use the term "context" in the sense of data compression, and not in the formal-language sense of "context-sensitive". It is however similar in that production rules now may vary according to the context in which they are. The context is determined by a context function, which only depends on the previous context and on terminal symbols. The authors suppose that in most applications, contexts are substrings seen in the past (on the left), so that context-dependent grammar permits overlapping parsing, with overlapped portions treated as contexts. Reflecting the first paper of the series, they define a set of grammar transformation, one algorithm and three coding schemas.

Unfortunately, an announced third part with complete results of implemented versions of these algorithms seems to have never been published and our attempts to contact the authors remained fruitless.

2.3.5 RNA compression with SLG

We will review and compare attempts of compressing DNA sequences with SLG below (Sect. 2.7.3). Here we will only consider compression of RNA sequences.

Liu et al. [149] present **RNACompress**, an algorithm that uses SLG to compress RNA sequences *and* the information that defines its secondary structure. It takes as input two sequences: s_{RNA} over alphabet $\{a, u, c, g\}$ contains the RNA nucleotides sequence and s_{SS} over $\{(,), \cdot\}$, with balanced brackets and $|s_{SS}| = |s_{RNA}|$.

It is different from the other algorithms we consider, because it does not make any inference of the grammar over the sequence. In fact, it uses the generic grammar G_{RNA}:

$$\begin{array}{lll} S & \rightarrow & NS|\epsilon \\ N & \rightarrow & aSu|uSa|cSg|gSc| \\ & & uSg|gSu|a|u|c|g \end{array}$$

Sequence s_{RNA} is compressed by indexing this rules from 1 to 12 and sending the indices of the rule that has to be applied in a derivation that generates the input sequence. However, the cleverness of the algorithm is that it uses the derivation that is specified by the left-most derivation of sequence s_{SS} with grammar G_{SS}:

$$\begin{array}{lll} S & \rightarrow & NS|\epsilon \\ N & \rightarrow & (S)|\cdot \end{array}$$

It is able to send both sequence inside one grammar by mapping the derivation tree of the left-most derivation of grammar G_{SS} of s_{SS} on s_{RNA}.

The decoder uses the structure of the received grammar to decode s_{SS} and the yield of the axiom to decode s_{RNA}. The indices for rules of G_{RNA} are encoded with a Huffman Code where the probability of each index is fixed and obtained by counting frequency of paired and unpaired bases in an RNA database.

2.3.6 XML Compression with SLG

Extensible Markup Language (XML) is the de-facto standard for representation of structured data on the World Wide Web. It is an extension of HTML, and permits to create semi-structured documents, interleaving the content with structural information. Because of its wide diffusion and its verbosity, the need of compressing this data emerged, and resulted in a large number of papers[4]. We refer to the survey of Bordese [36] and Sakr [207] for recent overviews of characteristics and classifications of them.

The structure of a XML document can be represented as a tree. The similarity to a SLG is close, and several XML-specific compressor have been presented that uses context-free grammar. *Exalt* [228] combines a XML parser with the grammar transform operation reported in the work of Kieffer and Yang to produce an irreducible grammar which is encoded with an adaptive arithmetic coder. *AXECHOP* [142, 143] uses a standard strategy in XML compressors by treating differently the structural and data part of the document. While it encodes the data with a BWT algorithm, it uses MPM (see Sect 2.6.6) with an adaptive arithmetic coder to compress the structure. *XSeq* [147] takes a similar approach, but compresses both data and structure with SEQUITUR (applying it on each of the stream separately). An interesting feature is its possibility of processing queries directly over the compressed file.

2.3.7 Compressed Data Structures

In our current information oriented society, generated data are generally stored in databases which are accessed frequently to answer queries. Because of the exponential growth of collection of data, the idea in this line of research is to compress the data for storage perennially and be able to access if without need of decompressing. Traditionally, data compression was used when a sender wanted to compress its message so that it can reach its receiver faster. Clearly, this is detrimental when the time spent to decompress the whole data set plus the time required by the query outbalances the storage space that can be saved.

The challenge is therefore to compress the database in a way that still permits to access it efficiently. In general, the two queries that are required are *access* and *pattern matching* or *find*. The access operation refers to random access over the string, or decompression of substrings, while the find operation returns the set of positions where a given pattern occurs. Here we will review briefly the main techniques that uses SLG. The literature in this subject is growing rapidly in the last years. For a recent review of advances in general of compressed data structures, see Hon, et al.'s keynote at CPM 2010 [114], the invited talk by Ferragina at ESA 2010 [87] or the classical review of Navarro and Mäkinen [169] (or Chiu, et al. [58] for an experimental comparison).

SLG seems to be a natural framework to achieve this task. Its correspondence between non-terminals and substrings, plus the possibility of "zooming" should make them suitable. The seminal paper of Kida et al. [126] gives a general framework (that includes SLG) and show how to perform pattern matching queries on it. Maruyama et al. [159] improve upon this with a approximation algorithm related to REPAIR, but that does not replace all occurrence of a selected

[4] see http://webdocs.cs.ualberta.ca/~gleighto/research/xml-comp.html for a list of related publications.

pair. Maruyama et al. [160] takes this a step further analyzing pattern matching for context-sensitive grammars. With respect to random access, first steps were made in Gąsieniec et al. [100] by showing how to visit consecutive symbols in constant time. Recently, Bille et al. [30] present a SLG that permit $\mathcal{O}(\log n)$ random access time. Claude and Navarro [68] show a self-index SLG that support both operations efficiently (in $\mathcal{O}(h \log m)$ and $\mathcal{O}(m(m+h) + h\,o) \log n)$, where h is the height of the parse tree, m the length of the pattern and o the number of times it appears). Claude et al. [69] present two compressed indexes based again on the RePair algorithm.

There has been a particular focus by some groups to study the *edit distance* problem of strings that are represented by straight-line grammars. See Hermelin et al. [112] for an overview and references.

2.4 Kolmogorov Complexity

The roots of Kolmogorov Complexity range over different fields and similar concepts have been discovered independently in different places[5]. In an informal way it can be defined as the amount of information — expressed in bits — contained in a single object. Standard probability theory, and therefore Shannon's notion of information, is useful when considering a set of events or sequences to which probability or information content has to be applied. Supposing a uniform source over an alphabet of size two, the probability of sequence 1011001001 and of sequence 0000000000 is the same, but intuitively the second one seems to contain less information, or be less random. One of the advantages of Kolmogorov Complexity is that it permits to express what a random sequence is: a sequence where the bits of information it contains is exactly the size of the sequence. A random sequence is a sequence from where no redundancy can be extracted.

For the definition, fix a Turing machine M

Definition 5 (Kolmogorov Complexity)**.** *The Kolmogorov Complexity of a string x is* $K_M(x) = \min\limits_{p:p \text{ is a program}} \{|p| : M(p) = x\}$

For a complete formal treatment of Kolmogorov Complexity see the classical reference of Li and Vitányi [145]. One of the main results that opens door to further work in Kolmogorov Complexity is the *Invariance Theorem* which says that for a universal Turing Machine M^*, $K_{M^*}(x) \leq K_M(x) + c_M$ for any M and c_M depends only on M. For as far as it concerns this thesis, it means that we can drop the subscript M, and talk of the information inside an object independently of the specific Turing Machine we use to express it. Another fundamental result is the following undecidability theorem. It can be proved with a self-referencing argument, similar to the Halting problem.

Theorem 2 (K is non-computable)**.** *There is no program p such that $p(x) = K(x)$, for any string x.*

A useful related definition is Conditional Kolmogorov Complexity $K(x|y)$ which is the size of the shortest program that outputs x on a universal Turing

[5] As mostly used in the literature we will talk of Kolmogorov Complexity, even if historically it would also be valid to talk of Solomonoff or Chaitin Complexity

machine if y is presented on a auxiliary tape. The relationship to conditional probability is evident. If $K(x|y) = K(x)$, then y does not add any information concerning x.

2.4.1 The SGP in Kolmogorov Complexity

A possibility of getting a computable approximation of Kolmogorov Complexity is to reduce the expressive power of the model from unrestricted grammars (recall from Sect. 2.1.2 that they are equivalent to Turing machines) to context-free.

In 1973, Klix [132] proposes, from a psychological point of view, a model that captures regularities (repetition of single elements, of pairs and mirroring) of a structured object. This model has a sequential representation and he proposes a measure of the content of the information of this model. He also presents results that show that the time that a human needs to learn by heart the object is linearly proportional to this measure. In the same proceedings, Scheidereiter [211] proposes to generate the sequence with a context-free grammar. He states the optimisation problem of finding the smallest grammar that generates the sequence (defining the size as the sum of the right-hand sides of the rules used in the derivation of the sequence). Scheidereiter states that "As, under relatively simple condition, there exists only a finite number of such grammars, one could find an optimal one by exhaustive search"[6], but proposes then an algorithm that selects long and frequent repeats (or reverse of a repeat), and uses this "local minimum" to "segment hierarchically" the sequence.

Ebeling and Jiménez-Montaño [80] propose in 1980 to apply this idea of using the size of a smallest context-free grammar that generates s as complexity measure of s on genetic sequence. The authors give small grammars for different protein, DNA and RNA sequences and compare this *grammar complexity* to other information measures. In this first paper, no algorithm for finding these grammars is presented and it seems that the resulting grammars are found one by one[7]. In a non-published paper from 1984 (see Ángel Jiménez-Montaño [9]) Jiménez-Montaño and Martinez seems to have defined the *Non-sequential Recursive Pair Substitution* algorithm, similar to RePair (see Sect. 2.6.7). Independently, the same definition of grammar complexity was analysed also by Charikar et al. [50] in 2002, where besides presenting an approximation algorithm, other related models are considered.

2.4.2 The Similarity Metric

Li et al. [146] proposes a practical application of Kolmogorov complexity for classification and clustering. They define a similarity between two sequences as:

Definition 6 (The Similarity Metric)**.**

$$d(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}$$

[6] "Da es unter relativ einfach Bedingungen nur endlich viele solcher Grammatiken gibt, könnte man durch Probieren eine optimale finden".

[7] For instance, for sequence (14) of length 126, a grammar of size 85 is given. Interestingly, one of the algorithms we are going to present (IRRMGP*, Sect. 4.3) finds a grammar of size 81.

The authors demonstrate that d is a "universal" metric (justifying the use of the definite article). Of course, as K is non-computable, for practical applications approximations are used. Any compression algorithm can be used as an approximation of Kolmogorov complexity and this is studied by Cilibrasi and Vitany [62, 63]. Given a compressor C, they define a distance between two sequences:

Definition 7 (Normalised Compression Distance)**.**

$$NCD_C(x,y) = \frac{|C(xy)| - \min(|C(x)|, |C(y)|)}{\max(|C(x)|, |C(y)|)}$$

By reducing the power of a universal Turing machine to a normal compressor the nice theoretical properties of universality and proper metric are lost, but the results they report, together with the freedom of parameter in such an approach, popularised this method. Other approximation of the Universal Metric are CD (Compression Dissimilarity) and UCD (Universal Compression Dissimilarity). See the general review by Ferragina, et al. [89].

Because of the hierarchical expression power of straight-line grammars, and the fact that they have been — since their re-discovery ten years ago — be connected to Kolmogorov complexity and structure discovery, it seems natural to use one of the straight-line grammar algorithms as a compressor in the NCD_C metric. Useful equations, like $|C(xy)| = |C(yx)|$, $|C(xx)| = |C(x)|$ and $|C(x|y)| = |C(x)|$ if y does not share any substring with x, hold (approximately) for the case that $C(x)$ is a smallest grammar of x. However, to our knowledge, only very recently such an approach was proposed by Cerra and Datcu [47]. For this, the authors use the grammars returned by RePair (see Sect 2.6.7), and define a complexity approximation of the sequence based on the number of rules of this grammar (each rule of a RePair has a right-hand size length of two). This is then successfully applied to cluster hierarchically mitochondrial DNA of different mammals and satellite images of different surfaces and vegetation.

2.5 Structure Discovery

Besides their compression capacity, the ability to infer a structure over the sequence has been traditionally the main motivation for work in SLG. The objective here is not to identify a target language, but to identify how the sequence is structured into segments, and how this segments are related to each other in an hierarchical manner.

To achieve this, we have to analyse the unique parse tree given by the resulting grammar. Because the non-terminals are anonymous and do not have any other meaning than to identify equal subsegments, the original grammar can be completely recovered from the parse tree or from the associated bracket set. Compared to a simple segmentation of the sequence, a parse tree has the advantage that it permits to zoom in and out, considering different level of segmentation, and that it reveals how bigger bricks are composed of smaller ones.

For a generic algorithm, Occam's Razor justifies the search for *small* grammars. If two grammars produce the same string, then this principle suggest that the smaller of both is more likely to be the correct one. In particular, a smallest grammar is one that achieves to extract all possible redundancy that can be expressed with a context-free parse tree.

One of the first to use a grammatical approach to segment hierarchically a sequence was Wolff [242], who in 1975 analysed the result of a computer program that replaces iteratively the most frequent pair of symbols by a new symbol.

Applications of SEQUITUR (see Fig. 2.3) are between the most known examples of examples of how the final grammar expose some of the real structure underlying the input sequence. Even if in almost all cases the examples are more anecdotic than quantitative, they present some interesting possible applications, ranging from natural language and musical structure identification to improving rendering performance and using it to easy keyword retrieval from a large data set.

In the same line, Evans [82] develops MDLCOMPRESS through the computation of the Symbol Compression Ratio (SCR), an heuristic to evaluate a symbols contribution to the overall sequence complexity. His OSCR algorithm — a previous version of MDLCOMPRESS — selects iteratively words that reduce best this amount. This is then applied to detect intruders in a framework of information system security. In Evans et al. [84] the constituents of the final grammar are used to detect binding sites of miRNAs which lead to tumorigenesis.

As we have seen before, most of the intuitive idea behind Kolmogorov Complexity is that, the smaller the description, the better it captures the real structure of the string. The idea of measuring the complexity of life became closer with the discovery that all the information transmitted to an offspring is encoded in a sequence, the DNA. Therefore, there have been many interesting attempts to define a measure of complexity of sequence, without complete success until now [39]. One of these attempts using SLG was performed by Lanctot et al. [138] who took as complexity of a DNA sequence an entropy estimator of the resulting grammar of their GTAC algorithm (see Sect. 2.6.8). In their results they were able to distinguish between coding (slightly higher entropy) and non-coding regions and observe that highly expressed genes have lower entropy then normal ones.

In Structure Discovery, the results are difficult to evaluate. Most of the work, like Wolff [242] or Nevill-Manning [171] show examples of the potential, but without a rigorous quantitative analysis of the quality of the structure found. Evans et al. [84] show interesting results but, at least on the validation on genetic sequences, the evaluation is performed on the constituents of the grammar. Not much is said about the hierarchical structure that is found.

2.6 Algorithms

In this section we will review existing algorithms that take a sequence as input and output a straight-line grammar that generates this sequence. We start with the LZ77 algorithm. While its output cannot be mapped directly to a context-free grammar, it holds strong links to the Smallest Grammar Problem

2.6.1 LZ77

LZ77 is maybe the most influential compression algorithm because of its simplicity, theoretical analysis and widespread use in commercial algorithms like *Deflate* used in Phil Katz' famous *PKZIP* tool and afterwards in *gzip* and the *png* file format.

The first work that described what would be known as the LZ77 algorithm by Ziv and Lempel [248] was rather theoretical, and several different implementations were given in the next years, usually adding the initial of the inventor after the traditional LZ. What we will describe here is not a real compression algorithm but a *factorisation* of the sequence [53].

LZ77 outputs a list that contains in each position a single character or a tuple off two integers. It processes the sequence online from left to right. At each position i, it looks for the longest prefix of $s[i..]$ that appears in $s[..i-1]$. If it is of length ℓ and appears at position m then LZ77 outputs the tuple $\langle m, \ell \rangle$. If none such substring exists, it outputs the character $s[i]$. It is important to note that the output cannot be trivially interpreted as a context-free grammar. This is because a pair $\langle m, \ell \rangle$ can refer to a substring that starts inside one pair, but ends after it. Even more, it is known result that the size of an LZ77 factorisation is a lower bound on the size of a smallest grammar. This is used by some of the approximation algorithms that ensure a worst-case approximation [51, 200].

2.6.2 LZ78

Defined by Lempel and Ziv in 1978 [249] as a successor of LZ77, its output can be mapped to a context-free grammar in Chomsky Normal Form. A variant named LZW is still used in the GIF image format and in the *compress* utility.

As in the description of LZ77, we will focus here on the most general abstract description, without entering the small but important details that make this algorithms run fast and efficiently in practice. In particular we will ignore any issues related to the size of the window and the size of the dictionary.

LZ78 reads the sequence symbol by symbol in an online way from left to right. It keeps a dictionary of words that is initialised with the empty string. At each position i it takes the index j of the longest word w in the dictionary such that $s[i : i + |w|] = w$, outputs the tuple $\langle j, s[i + |w|] \rangle$ and adds the word $s[i : i + |w| + 1]$ to the dictionary.

If interpreted as a context-free grammar, the final grammar is always in CNF. Each right-hand side consists of a non-terminal (an index from the dictionary) and the terminal character that did not permit a longer match.

2.6.3 Sequitur

Sequitur is probably the most popular of existing algorithms that infer a straight-line grammar. His popularity is due to its efficiency (linear in the size of the sequence), its timely appearance and successful application to model sequences of diverse origin.

Sequitur also processes the sequences from left to right reading one character at a time. It maintains two invariants: every digram appears only once in the grammar ("digram uniqueness") and every rule is used at least once ("rule utility"). After reading each symbol c, it is appended to the axiom rule (initialised

The bibliography over SEQUITUR is scattered over several publications, so what follows is a short summary in chronological order. The algorithm is first mentioned in Nevill-Manning et al. [178], where general characteristics of straight-line grammars are exploited for compression. This is then applied for compressing structured data in form of a genealogical database [179]. The generated grammar is generalised manually and automatically, looking at patterns on the final grammar (see Sect. 1.2.1). The thesis of Nevill-Manning [171] dates from 1997 and is probably the most exhaustive description of SEQUITUR. In Nevill-Manning and Witten [174] the authors extend the conclusion of the thesis and analyses further its compression capacity. [173] analyses the time complexity of SEQUITUR and the size of the grammars it generates with respect to the size of the original sequence. Nevill-Manning and Witten [175] discuss the possibility of complementing a straight-line learning process with a more traditional inference process that supports branching and looping. The probably most compact reference for SEQUITUR is [172]. The linear memory usage is further analysed in [176] and various methods are presented to permit the algorithm to run in bounded space. Nevill-Manning et al. [180, 181] present a software that permits to browse a huge collection of library items. These papers also present some drawbacks of the way SEQUITUR defines lexical significant constituents. Finally, Nevill-Manning and Witten [177] compare the size of the resulting grammar of SEQUITUR with other offline algorithms.

Figure 2.3: A bibliographic overview of SEQUITUR

with the empty string). If the digram formed with its predecessor symbol C and c appears exactly as a right-hand side of another rule ($N \rightarrow Cc$), the digram is replaced by N. If instead Cc already appeared before, a new rule $N \rightarrow Cc$ is created and both occurrences of Cc replaced by N. Such a replacement has as consequence a reduction in the number of occurrences of C which can produce that it does now appear only once. If C is a non-terminal, this violate the second constraint and therefore the rule $C \rightarrow \alpha$ is eliminated and the only occurrence of C replaced by α.

Thanks to a neat algorithmic design, and supposing constant-time look-up in a hash-table, the run time of SEQUITUR is linear. See Figure 2.3 for an overview of the existent literature presenting SEQUITUR.

2.6.4 DNASequitur

Trying to compress DNA through a context-free grammar, DNASEQUITUR [56] modifies SEQUITUR and analyses different transformations from a SLG to a symbol stream. The modifications to SEQUITUR consist in considering also reverse complements. Beside this, the main features of the algorithm (the online behavior and the two constraints) remain unchanged.

2.6.5 Sequential

An undesirable property of SEQUITUR is that two non-terminals may produce the same constituent. An algorithm presented by Kieffer and Yang [127] and later called SEQUENTIAL by Charikar et al. [51] addresses this issue and modifies SEQUITUR by adding a third constraint that ensures that this will not happen. While producing smaller grammars in the general case, this results in a non-linear algorithm.

2.6.6 Bisection / MPM

Suppose that n, the size of the sequence, is a power of 2. The BISECTION algorithm, outlined in Kieffer and Yang [127], splits the sequence iteratively in two and assigns the same non-terminal if the substring is the same. For the case that n is not a power of 2, the axiom right-hand side consists in two or more non-terminals of decreasing powers of two. This was then generalised in Kieffer et al. [128] with the MULTILEVEL PATTERN MATCHING (MPM) algorithm, permitting to split the sequence into more than two parts.

2.6.7 RePair

Instead of an online treatment of the sequence like the LZ family or SEQUITUR, REPAIR by Larsson and Moffat [139] takes an off-line approach and considers all digrams of the original sequence. Each iteration consists in replacing the most frequent digram of the current sequence by a new symbol. The fact of focusing only on digrams permits a linear implementation which is given in the same paper. A theoretical analysis of its compression capacity can be found in Navarro and Russo [170]. It should be noted that REPAIR was not the first to implement this idea. Wolff [242] used it for pattern discovery in natural language and Ángel Jiménez-Montaño [9] for an approximation of his grammar complexity on genetic sequences. In pure data compression, the *Byte-pair encoding* compression algorithm implements this idea [92]. See also Bell et al. [26, Chapter 8.2.1] and corresponding bibliographical references.

2.6.8 Longest First

A similar — but somehow opposite — idea of REPAIR is to select the *longest* repeat in each iteration. The core idea was introduced in 1999 [27] (in the same conference and the same session as REPAIR) as a general purpose compressor. It is also an off-line algorithm and looks for interesting repeats of the original sequence. The algorithm iteratively selects the longest repeat in the sequence, extracts it and replaces all the occurrences of this repeat with a pointer. Later this idea was extended in order to also take into account the right-hand side of previous introduced rules. In Lanctot et al. [138] this heuristic is used to define the algorithm GTAC which is an entropy estimator of DNA sequences. For a long time, claims of a linear implementation for this algorithm were made [116, 138, 167, 177] and finally such an algorithm was given by Nakamura et al. [168], based on sparse lazy suffix trees.

2.6.9 Greedy

Similar to LongestFirst and RePair, Greedy [13] is another off-line algorithm that selects in each iteration a repeat to be replaced by a new symbol. In this case, the repeat that would yield the best contraction of the final grammar is chosen. Apostolico and Lonardi [13] define a compression schema and the contraction caused by a word is defined accordingly. This is applied particularly to biological sequences [14]. Nevill-Manning and Witten [177] use the final size of the grammar as size measure, and their algorithm Compressive selects a repeat whose replacement reduces the most the size of the grammar.

2.6.10 MDLCompress

MDLCompress [82, 84] is similar to the strategy of Greedy and Compressive. It selects in each iteration a repeat that reduces the most the description length of the grammar. The motivation however, consists in finding a correct model rather than compression. The authors interpret a SLG as a model (the axiom rule) plus data (the rest of the rules) and derives from there a more sophisticated score function. The resulting grammar is used less for its compression capacity than for its potential to detect intruders and discover MicroRNA targets (see Sect. 2.5). The original algorithm [83], only considered occurrences in the axiom rule, but MDLCompress also considers the "model" and changes the score function used to select a repeat. This score ("symbol compression ration") of word s is defined as:

$$\frac{|pos_s(w)|\left(\log_2\left(\hat{R}\right) - \log_2\left(|pos_s\left(w\right)|\right)\right) + |w|}{|pos_s(w)| * |w|}$$

with $\hat{R}$ "constant for a given partition of sybmols". Evans et al. [84] also make references to some post-processing particular to DNA sequences, but no details are given.

2.6.11 Bounding the Worst Case

Since the re-discovery of smallest grammars as approximation for Kolmogorov complexity by the work of Lehman, Charikar and co-workers [51, 141] several papers define algorithms that ensure that the ratio of the size of a smallest grammar g^* and the size of the resulting grammar g is bounded in the worst case.

Charikar et al. [51] themselves conclude giving two algorithms that achieve better worst case approximations than the bounds they found for other algorithms. The first is based on an approximation algorithm for the shortest superstring problem with an approximation ratio of $\mathcal{O}(\log^3 n)$. The second algorithm achieves an approximation ratio of $\mathcal{O}(\log n/g^*)$, and is based on the LZ77 factorisation of the original sequence and involves a rather complicated maintenance of a balanced binary tree. This is simplified in a simultaneous work by Rytter [200] that achieves the same ratio using also the LZ77 factorisation and an AVL-tree.

After this initial approximation, Sakamoto continued the work. Based on the RePair algorithm, he presents in [204] a linear algorithm that achieves a

$\mathcal{O}(\log^2(n))$ approximation ratio, which was improved in a journal version [205] to $\mathcal{O}(\log n/g^*)$. The next challenge he and his co-workers focused on was to reduce the space requirements and they present [206] a linear algorithm that only needs $\mathcal{O}(g^* \log g^*)$ space with an approximation ration of $\mathcal{O}(\log g^* \log n)$. Recently they presented yet another algorithm that performs well on all three fronts: it consumes $\mathcal{O}(g^* \log g)$ space, needs $\mathcal{O}(n \log^* n)$[8] time and achieves an $\mathcal{O}((\log^* n) \log n)$ approximation ration. Another variant is proposed by Gagie and Gawrychowski [93], where they consider a streaming model and prove that with constant memory and a logarithmic number of passes over a constant number of streams, a $\mathcal{O}(\min(g \log g, \sqrt{n/\log n}))$ approximation algorithm is possible.

It should be noted that the definition of size of a grammar in these papers differs from ours (Def. 2), because there $|G|$ is defined as being the sum of the symbols in the right-hand side only (we will denote by m this amount). This is practical for an asymptotic behaviour because it eases the calculations and because the number of non-terminals is bounded by $m/2$. A similar argument can be used to justify the use of grammars in Chomsky Normal Form in most of these algorithms: as they are no unitarian nor ϵ-rules, the ratio of the size of the canonical CNF and the size of the original one is constant. However, these constants can make a huge difference in practical applications (see Sect. 2.7).

2.7 Comparison

In the presentation of the algorithms in the previous section, we observe an evolution from on-line algorithms (LZ78, Sequitur and its descendants) to linear off-line (RePair, LongestFirst) to reach finally more complex off-line algorithms (Greedy, MDLCompress). In this section we analyse the consequences that this increasing complexity of these algorithms has on the size of the final grammar.

Previous studies performed similar comparisons, but considered the asymptotic lower and upper bound for a worst case. As we have seen, the standard measure is the ratio between the size of the output grammar, and the size of a smallest grammar. In particular, Charikar et al. [51] analyse and compare the approximation ratio of existing algorithms, including LZ78, Bisection, Sequential, LongestFirst, Greedy and RePair. Of these, the one with best upper bound is Bisection with $\mathcal{O}((n/\log n)^{\frac{1}{2}})$. However, it is not resolved if these bounds are tight. Considering the difference with respect to the known lower bound this does not seem to be the case. LongestFirst, Greedy and RePair are upper-bounded in general and the best known lower bound for the worst case for Greedy is a constant $(5 \log 3/(3 \log 5) \approx 1.138)$.

Also, the use of the Big-O notation in this analysis can be misleading in practical applications. It is known [51] that the size of the LZ77 factorisation of a sequence is a lower bound on the size of a smallest grammar for this sequence, which is the reason that the best approximation algorithms are based on this decomposition. We computed the LZ77 factorisation on the Canterbury corpus (see Table 2.2(a)). For all but one file (namely, `ptt5`) the size of the LZ77 decomposition is bigger than $n/\log_e(n)$, which means that the trivial grammar

[8] $\log^* n$ is the iterated logarithm of n, the maximal number of logarithms in the expression $\log\log ... \log n$, such that it is greater than 1.

$\langle \Sigma, \Sigma \cup \{S\}, \{S \to s\}, S\rangle$ of size $n+1$ is already within an $\log n$ factor of a smallest grammar for all but one sequences of the Canterbury corpus. As often, the constant factor hidden in the Big-O notation can have dramatic consequences in practice.

Regarding practical application, a similar — but much more reduced in scope — comparison was performed by Nevill-Manning and Witten [177].

2.7.1 IRR: a general offline framework

As we have seen, most offline algorithms follow the same general scheme. First, the grammar is initialised with a unique initial rule $S \to s$ where s is the input sequence and then they proceed iteratively. At each iteration, a word ω occurring more than once in s is chosen according to a score function f, all the (non-overlapping) occurrences of ω in the grammar are replaced by a new non-terminal N and a new rewriting rule $N \to \omega$ is added to the grammar. We give pseudo-code for this general scheme that we name **Iterative Repeat Replacement (IRR)** in Algorithm 1. Recall that $\hat{\mathcal{R}}(s)$ is the set of non-overlapping repeats of size at least two and $\mathcal{L}_\omega(s)$ the normalised non-overlapping list of occurrences of ω in s. $G_{\omega \mapsto N}$ is the result of replacing each occurrence of ω in $\mathcal{L}_\omega(r(G))$ by a new symbol N and adding the rule $N \to \omega$ to the set of productions.

Algorithm 1 Iterative Repeat Replacement (IRR)

IRR(s, f)

Input: s is a sequence, and f is a score function

```
1: 𝒢 ↤ G({N_0 → s})
2: while ∃ω : ω ↤ argmax_{α∈R̂(r(G))} f(α, G) ∧ |G_{α↦N}| < |G| do
3:     G ↤ G_{ω↦N}
4: end while
5: return G
```

The IRR scheme enables us to compare in a uniform framework the behaviour of different score functions f that are used in the classical algorithms for choosing the words to replace. LongestFirst correspond to $f(\omega, G) = f_{ML}(\omega, G) = |\omega|$. Choosing the most frequent repeat, like in RePair, corresponds to use $f(\omega, G) = f_{MF}(\omega, G) = |\mathcal{L}_{r(G)}(\omega)|$. Note however the difference that IRR is more general than RePair and may select a word which is not a digram.

In order to derive a score function corresponding to Compressive, note that replacing a word ω by a non-terminal results in a contraction of the grammar of $(|\omega|-1) * |\mathcal{L}_{r(G)}(\omega)|$ and its inclusion in the grammar adds $|\omega|+1$ to the grammar size. This defines $f(\omega, G) = f_{MC}(\omega, G) = (|\omega| - 1) * (|\mathcal{L}_{r(G)}(\omega)| - 1) - 2$. We call these three algorithms IRR-ML (maximal length), IRR-MF (most frequent) and IRR-MC (maximal compression), respectively.

The complexity of IRR when it uses one of these scores is $\mathcal{O}(n^3)$: for a sequence of size n, the computation of the scores involving only $|\mathcal{L}_{r(G)}(\omega)|$ and $|\omega|$ of the $\mathcal{O}(n^2)$ possible repeats can be done in $\mathcal{O}(n^2)$ using a suffix-tree–like structure. The number of iterations is bounded by n since the size of the grammar decreases at each step.

2.7.2 Final grammar size

We performed a comparison of the final grammar size obtained by the algorithms presented in Sect. 2.6 on the Canterbury and DNA corpora (see Appendix A). We exclude MDLCompress and Greedy whose goal is to compress the final bitstream (also, no public version of MDLCompress is available, only source code for the older OSCR algorithm [82]). We could not found any available implementation of DNASequitur[9] and it is not clear how to weight reverse-complement non-terminals when comparing the grammar size. For Bisection and Sequential we used Yann Ponty's implementation[10]. It presented problems with the files containing non-printable symbols of the Canterbury corpus, so we excluded these files for these two algorithms. José Rondo from the University of Chile implemented the approximation algorithm of Rytter [200] (personal communication). As the final grammars is completely in Chomsky Normal Form, we only report the sum of the right-hand side (not the number of productions). Again, the implementation presents some problems with the file containing non-printable symbols which we thus excluded. For LongestFirst, RePair and Compressive we used our own IRR implementation For LZ78 we post-processed the output factorisations to transform them into context-free grammars, where every non-terminal appears at least twice (if not, it is eliminated and replaced with its right-hand side). Finally, the reader should bear in mind that a tuple $\langle m, \ell\rangle$ counts as one for the computation of the size of the LZ77 factorisation.

The results in Table 2.2 reveal the preeminence of the greedy strategy of IRR-MC. IRR-MF comes close (obtaining a smaller grammar in one case) and could be interesting because of the linear RePair and the additional useful information that every rule length is of size two. But the huge difference in the case of sequence `humghcs` — a sequence with high number of repeats (see Table A.3) —, where the grammar obtained by IRR-MF is a 25% bigger than the one of IRR-MC illustrate that there are cases where the final size can vary drastically. Bisection and LZ78 perform poorly, but this seems obvious because it is not their goal to optimise the size of the final grammar.

2.7.3 DNA Compression

Thanks to the easily interpreted structure they generate and the failure of other general purpose algorithms, it seemed natural to apply SLG to compress DNA sequences. To our knowledge, there have been four different attempts:

1. Greedy (2000, see Sect. 2.6.9) [14],
2. GTAC (2000, see Sect. 2.6.8) [138],
3. DNASequitur (2004, see Sect. 2.6.4) [56],
4. MDLCompress (2007, see Sect. 2.6.10) [84].

We summarised the resulting compression in Table 2.3. The results for the Greedy algorithm [14] were obtained with the software the authors published[11]. GTAC [138] only reports a theoretical entropy measure, not the real

[9] The original code of N. Cherniavsky got lost (personal communication)
[10] http://yann.ponty.free.fr/approximations.html
[11] http://www.cs.ucr.edu/~stelo/Offline/

Table 2.2: Final grammar size for straight-line grammar algorithms on Canterbury (a) and DNA corpus (b). Absolute numbers are given for LZ77 and IRR-MC only, the others are given as percentage with respect to IRR-MC. For Rytter only the sum of the length of the right-hand sides is given, as the final grammar is in CNF. The best for each row is boldfaced.

(a) Canterbury Corpus

sequence	LZ77	Rytter	Sequential	Sequitur	Bisection	LZ78	IRR-ML	IRR-MF	IRR-MC
alice29.txt	22,906	250.54	14.08	19.87	236.89	105.71	36.72	3.54	**41,000**
asyoulik.txt	21,643	249.41	14.60	17.74	229.34	95.23	37.35	2.76	**37,474**
cp.html	4,587	178.88	28.73	22.20	248.46	102.40	19.43	5.36	**8,048**
fields.c	1,871	171.55	41.54	20.26	297.01	136.39	16.51	10.22	**3,416**
grammar.lsp	855	160.56	41.68	20.16	255.67	102.04	17.45	9.64	**1,473**
kennedy.xls	152,224	–	–	4.40	–	28.67	7.69	0.09	**166,924**
lcet10.txt	52,611	303.28	18.97	24.54	273.66	169.90	44.74	3.12	**90,099**
plrabn12.txt	72,628	284.51	7.15	14.86	219.96	70.35	45.09	0.94	**124,198**
ptt5	25,467	–	–	23.39	–	66.98	25.07	1.12	**45,135**
sum	7,914	–	–	25.31	–	99.24	13.59	6.21	**12,207**
xargs.1	1,172	147.01	25.62	16.10	230.71	82.75	12.36	6.53	**2,006**
average	–	218.22	24.05	18.98	248.96	96.33	25.09	4.50	–

(b) DNA Corpus

sequence	LZ77	Rytter	Sequential	Sequitur	Bisection	LZ78	IRR-ML	IRR-MF	IRR-MC
chmpxx	16,458	302.58	3.62	5.61	167.94	42.4	59.35	0.01	**28,706**
chntxx	21,604	306.63	2.83	5.93	174.29	41.25	58.88	0.03	**37,885**
hehcmv	31,085	291.36	3.63	4.67	178.94	43.25	61.09	0.09	**53,696**
humdyst	6,143	271.77	3.46	5.92	160.28	37.48	53.29	0.02	**11,066**
humghcs	9,945	264.93	46.36	20.3	250.92	91.86	36.32	25.46	**12,933**
humhbb	10,894	279.61	7.99	7.16	176.20	44.38	54.72	2.27	**18,705**
humhdab	8,928	273.46	6.42	9.77	169.64	44.99	51.74	0.27	**15,327**
humprtb	8,622	272.68	5.47	7.74	169.96	43.98	52.94	0.35	**14,890**
mpomtcg	25,774	310.05	5.08	5.62	182.07	44.12	59.01	0.90	**44,178**
mtpacga	14,060	300.70	4.51	6.05	169.52	42.46	57.00	0.29	**24,555**
vaccg	25,718	303.81	3.17	5.37	177.56	46.13	61.62	**-0.05**	**43,701**
average	–	288.87	8.41	7.65	179.76	47.48	55.09	2.69	–

sequence	DNA SEQUITUR	IRR^c-ML	GREEDY	MDL COMPRESS	AAC-2	IRR^c-ML-5
chmpxx	2.12	3.1635	1.9022	-	1.8364	1.6929
chntxx	2.12	3.0684	1.9986	1.95	1.9333	1.6306
hehcmv	2.12	3.8455	2.0158	-	1.9647	1.8765
humdyst	2.16	4.3197	2.3747	1.95	1.9235	2.2396
humghcs	1.75	2.2845	1.5994	1.49	1.9377	1.9626
humhbb	2.05	3.4902	1.9698	1.92	1.9176	1.9278
humhdab	2.12	3.4585	1.9742	1.92	1.9422	1.9913
humpr	2.14	3.5302	1.9840	1.92	1.9283	1.9682
mpomtcg	2.12	3.7140	1.9867	-	1.9654	1.9737
mtpacga	-	3.4955	1.9155	-	1.8723	1.8767
vaccg	2.01	3.4782	1.9073	-	1.9040	1.7861

Table 2.3: Results of existing SLG compressors that have been applied to DNA sequences. IRR^c refers to the IRR algorithm where the complimentary strand is also taken into account. All numbers refer to bits per symbol.

bit string[12]. We used our own implementation of LONGESTFIRST (IRR-ML), adding an option of searching also for complimentary repeats (like in the original GTAC algorithm). We encoded $r(G)$ with a 0-order adaptive arithmetic coder, adding one bit per non-terminal to differentiate normal repeats from reverse-complement ones. Unfortunately, copyright issues prevent a public available version of MDLCOMPRESS. Our results including the published score function in our IRR schema yielded slightly different results which may be due to the post-processing steps applied by the authors or just on how ties were resolved. We preferred therefore to report in Table 2.3 only published results for this algorithm [84].

For the sake of comparison, we completed this with the results using an higher-order adaptive arithmetic coder with a context of 2 (which is reported by Grumbach and Tahi [107] to be the value that achieves best compression). The results of IRR-ML are surprisingly bad. We suppose that is due to the big number of rules this algorithm generates, and to compare we run IRR-ML for only five iterations. The result can be appreciated in the last column and should be compared with the result of state-of-the-art DNA compressor (Table 2.1). It performs well in general, but some exceptions (`humdyst` and `humghcs`) reveal that a more elaborated schema is necessary to yield a competitive DNA compressor. The only algorithm that outperforms (even if only slightly) AAC-2 is MDLCOMPRESS, excepting one case.

We conclude this comparative study remarking that the additional complexity of off-line algorithms pays out in the final size. For DNA Compression, the best results are obtained with MDLCOMPRESS, even if none of them seems to be able to compete against standard DNA compressors. The difference is even more acute if the comparison is performed regarding the (somehow more direct measure of) size of the final grammar. The off-line algorithms perform much better than any of the on-line ones. Inside those, IRR-MC itself differentiates

[12] Also, Nakamura et al. [168] report an error in this algorithm, which leads (in the best case) to a non-linear algorithm or to an erroneous output

clearly from the others and represents the current state of the art.

Chapter 3

Efficiency

We have seen a wide range of algorithms that infer a straight-line grammar from a given sequence. Some of them are on-line and strive for speed and reduced space use. They are adapted to work efficiently in on-line applications, streaming pipelines or in general for use where time or memory are very reduced. However, when compared with respect to the size of the final grammar, the metric we use to measure the quality of the final structure, they perform poorly compared to off-line algorithms. Moreover, when the final goal is to learn something about the structure, time is less constraining. Of course, the final algorithm still has to be feasible.

Therefore, in this section we consider the general IRR framework. We experimented with different score functions to choose the right constituents, and the MC strategy proved to obtain the best results. Here, we will focus on how to implement efficiently the off-line IRR framework in a way that permits it to scale easily.

There has been considerable work to improve the efficiency of individual IRR algorithms. In particular, we already mentioned that RePair [139] – which is similar to IRR-MF – runs in provable linear time in the size of the input. The same is true for LongestFirst – equivalent to IRR-ML – based on a careful update of a sparse lazy suffix-tree [168]. These solutions however are ad-hoc and depend strongly on special characteristics of this strategy. For example, any repeat selected by IRR-MF will never appear inside a right-hand side of a non-axiom rule. Conversely, any selected repeat by IRR-ML will never contain a previously introduced non-terminal. Other choices for the function score violated these invariants and it is not clear how to adapt the given solutions to the general case.

The bottleneck (both asymptotically and in practical instances) in these offline algorithms lies in computing all repeats and calculating the score function for every such substring. This presents a real problem when these algorithms are being applied on DNA sequences: not only can the sequences be longer by orders of magnitude, but the presence of long repeats means that the number of repeats can grow very fast (see Fig. 3.1(a)). Most of these repeats may not be interesting, in the sense that their score is known to be smaller than that of another. This reasoning leads to a definition of equivalence classes and maximality of a repeat. In Sect. 3.2 we consider different classes of repeats and analyse how the IRR algorithms behave if the repeats they consider are limited to these classes. For one of these classes we present a linear algorithm to compute all repeats of this class.

In Sect. 3.3 we consider the effects on the final grammar size and execution time of not filtering out overlapping repeats, and estimate the number of non-overlapping occurrences with the total number of occurrences.

Finally, we consider how to reduce the time spent in computing all repeats by using the information that was computed in the previous iterations. Most of the repeats are not modified from one iteration to the next, and at an intuitive level it seems that much computation cycles are wasted calculating the same result over and over again. A similar approach was considered by Markham et al. [158] in order to improve the efficiency of their MDLCompress algorithm. They store in a table the information necessary to compute the score for all repeats, together with pointers that permit to update this information. Though the authors report that the number of repeats apparently is linear in the sequence they tried, a theoretical upper bound is $\mathcal{O}(n^2)$. Our solution differs from this, and is inspired by the fact that most of the time is spent in computing the possible repeats, and not in the computation of the score once the repeats are given. We present therefore in Sect. 3.4 a data structure almost equivalent to an enhanced suffix array, that permits to be updated efficiently while a repeat in the index sequence is replaced by a new symbol.

A preliminary version of Sect. 3.2.3 and Sect. 3.3 was realised in collaboration with the Natural Language Processing Group from the University of Córdoba, Argentina through a joined INRIA/MINCyT project and accepted for publication in 2010 [46]. Sect. 3.4 was realised in collaboration with Pierre Peterlongo, from the Symbiose team at the INRIA research center of Rennes, and was published in 2009 [96], based on a preliminary version presented at the *Prague Stringoloy Club* conference in 2008 [95].

For all these algorithmic improvements, we will use an index data structure called *enhanced suffix array* which is defined in the following section.

3.1 The Suffix Array

A suffix array is part of the suffix-tree data structure family. It consists in a lexicographically ordered array of all suffixes of the input sequence. The suffixes themselves are not stored but instead their starting positions.

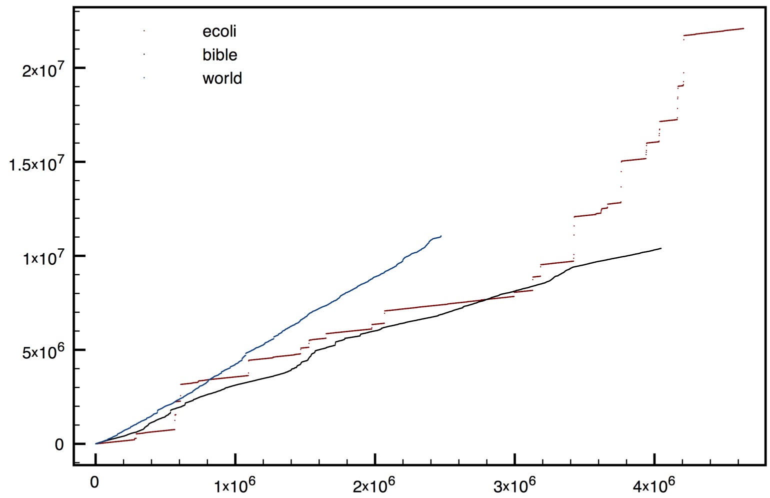
Definition 8 (Suffix Array)**.** *Consider a sequence s of length n over an alphabet Σ with an order $\prec$. The lexicographical extension to Σ^* will also be denoted by $\prec$. Let $\tilde{s} = s\$$, with a special character $\$$ not contained in Σ, smaller than every element of Σ.*

The suffix array, denoted by sa, is a permutation of $[0..n]$ such that:

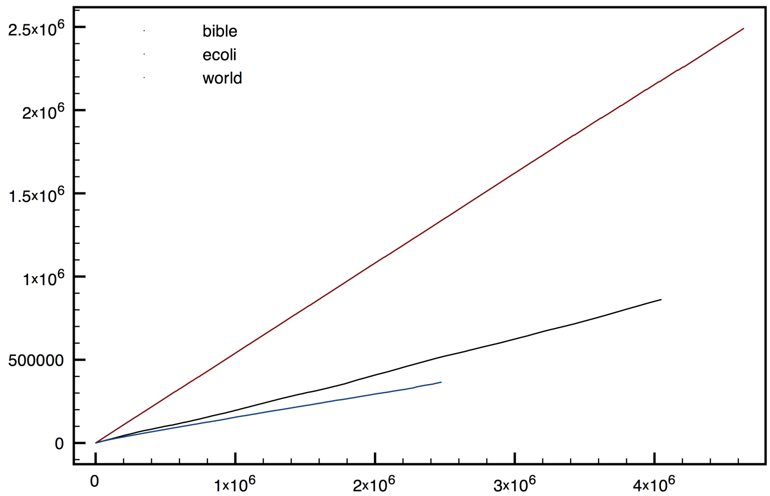
$$\forall\ i,\ 0 < i \leq n\ :\ \tilde{s}[sa[i-1]..] \prec \tilde{s}[sa[i]..]$$

Recall that $s[i..]$ denotes the suffix of s starting at position i.

Usually, the suffix array is used conjointly with an array called lcp, that gives the length of the longest common prefix between two suffixes whose starting



(a)



(b)

Figure 3.1: Number of repeats (a) and maximal repeats (b) over prefixes of the Large Corpus. Largest maximal and super-maximal repeats follow a similar trend to maximal repeats

positions are adjacent in sa. Formally,

$$\begin{aligned} & lcp[0] = 0, \\ \forall\ i \in [1, n]\ :\ & lcp[i] = k \textit{ such that} \\ & \tilde{s}[sa[i-1]..][..k-1] = \tilde{s}[sa[i]..][..k-1] \textit{ and } \tilde{s}[sa[i-1]..][k] \neq \tilde{s}[sa[i]..][k]. \end{aligned}$$

Eventually, a third array called isa (for inverse suffix array) may be used conjointly with sa and lcp. This array gives, for a position p in s, the index i in sa such that $sa[i] = p$*. Thus* $sa[isa[p]] = p$*.*

Suffix arrays can be constructed in linear time [123, 129, 133] but non-linear algorithms [140, 156] are usually more efficient for practical applications [192].

The union of *sa*, *lcp* and *isa* arrays is called an **Enhanced Suffix Array** (*ESA*). Enhanced suffix arrays are known to be equivalent to suffix trees [2] in the sense that they can easily be used to mimic a suffix tree. There are however more space efficient than suffix trees. This space improvement is compensated in general by an extra $\log n$ time factor when some kind of exact pattern matching has to be done: while this can be done straightforward on a suffix-tree by reading the word down the tree (in time $\mathcal{O}(m)$, with m the length of the pattern), on a suffix array a binary search is needed (thus $\mathcal{O}(m \log n)$). Alternatively, but using more extra arrays, this can be done in $\mathcal{O}(m+\log n)$ [153] or even $\mathcal{O}(m)$ [2].

To avoid confusion, we will use the term *position* when referring to the index over a sequence and *index* when referring to any of the arrays of an ESA.

3.2 A Taxonomy of Repeats

Maximal repeats appear in the literature (see notably the classical book of Gusfield [110]) as a compact representation of all repeats. Differently from normal repeats, the number of maximal repeats inside a sequence is linear and it is trivial to recover all repeats from the set of maximal repeats.

A maximal repeat is a repeat such that if it would be extended to its left or right it would lose some of its occurrences. For the definition, we will prefer the use of a set notation to refer to occurrences of repeat. $Pos_s(w) = \{\{j \in \mathbb{N} : i \leq j < i + |w|\} : i \in pos_s(w)\}$, the set of intervals of occurrences (where each interval is expressed as the set of positions). If w has length three, and appears in position 4 and 6 on sequence s (they overlap), then $Pos_s(w) = \{\{4, 5, 6\}, \{6, 7, 8\}\}$.

Formally:

Definition 9 (Maximal Repeats)**.** *The set of* **maximal repeats** *($\mathcal{MR}$) is the set of repeats such that:*
$\mathcal{MR}(s) = \{w \in \mathcal{R}(s) : \nexists w' \in \mathcal{R}(s) : \forall o \in Pos_s(w) : \forall o' \in Pos_s(w') : o \nsubseteq o'\}$

The property of maximality is strongly related to the context of a repeat. If the symbol to the left (right) of any occurrence of w is always the same, then w is not a maximal repeat because it could be extended to its left (right) without losing any occurrence.

A stronger property is super-maximality. A repeat is *super-maximal* if it is not a substring of any other repeat. They can then be defined by looking at the set of repeats alone, without need of referring to their occurrences. Our[1] definition is equivalent, but uses the notion of occurrences to mirror Def. 9.

Definition 10 (Super-maximal Repeats)**.** *The set of* **super-maximal repeats** *($\mathcal{SMR}$) is the set of repeats such that :*

$$\begin{aligned}\mathcal{SMR}(s) &= \{w \in \mathcal{R}(s) : \nexists w' \in \mathcal{R}(s) : \exists o \in Pos_s(w) : \forall o' \in Pos_s(w') : o \not\subseteq o'\} \\ &= \{w \in \mathcal{R}(s) : \forall w' \in \mathcal{R}(s) : \nexists o \in Pos_s(w) : \forall o' \in Pos_s(w') : o \not\subseteq o'\}\end{aligned}$$

Frequent repeats are more probable (supposing an i.i.d. source) to be maximal repeat than longer one, because only one different context suffices to define them as maximal. This implies that small repeats are likely to be maximal repeats. For super-maximality the opposite is true: small repeats are less likely to be super-maximal, and only long repeat have a chance of not being contained in another repeat.

Largest-maximal repeats lie in between. These are those repeats that have at least one occurrence not covered by another repeat.

Definition 11 (Largest-maximal Repeats)**.** *The set of* **largest-maximal repeats** *($\mathcal{LMR}$) is the set of repeats such that :*

$$\mathcal{LMR}(s) = \{w \in \mathcal{R}(s) : \exists w' \in \mathcal{R} : \nexists o \in Pos_s(w) : \forall o' \in Pos_s(w') : o \not\subseteq o'\}$$

Largest maximal repeats cover the whole sequence, which is not necessarily true for super-maximal repeats. But they do it in a less redundant way than maximal repeats. Gusfield names this set *near-supermaximal repeats* and uses them to facilitate the explanation of super-maximal repeats. The characterisation of the largest maximal repeat of Gusfield [110, Theorem 7.12.4, page 147] is given in terms of the leaves of the suffix tree associated to a sequence. Recently, this class of repeats were re-discovered [183] and successfully applied to the automatic detection of CRISPRs, a genomic structured found in archaea and bacteria that are expected to have a role in their adaptive immunity [199]. We will see in Sect. 5.4.1 that Definition 11 corresponds naturally to irreducible motifs when referring to rigid patterns.

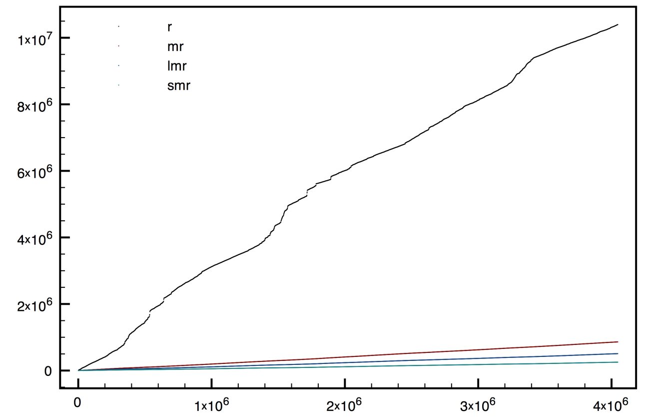
In Fig. 3.1 the difference between the number of simple and maximal repeats can be observed. Largest and super-maximal repeats follow a similar trend to Fig. 3.1(b).

3.2.1 Bounds

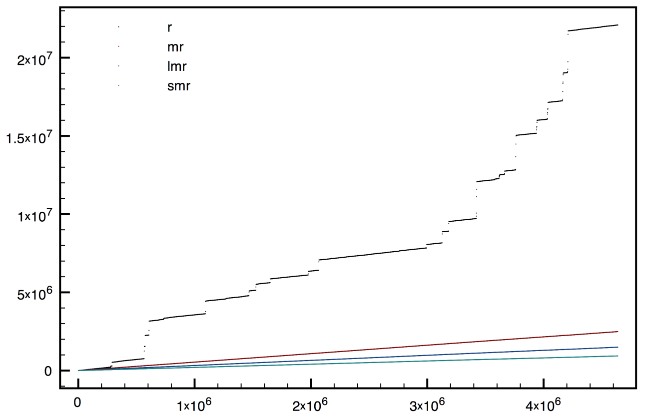
One of the key features of maximal repeats is that their total number can be bounded by the size of the sequence [110]. However, the total number of occurrences of maximal repeats can still be quadratic (see Lemma 3). Largest-maximal repeat could fill the space between the quadratic number of total occurrences of maximal repeats and the linear number of total occurrences of super-maximal repeats, but a tight upper bound is still unknown. Table 3.1 resumes what is known about the bounds of the class of repeats we presented. We proved all non-trivial bounds. $n_X(k)$ denotes $max_{s:|s|=k}\{|X(s)|\}$ where $\mathcal{X}$ stands for one of $\mathcal{R}, \mathcal{MR}, \mathcal{LMR}$ or $\mathcal{SMR}$ and $Occs_X(k) = max_{s:|s|=k}\{\sum_{w \in X(s)} |pos_s(w)|\}$.

Lemma 1. *$n_{\mathcal{X}}(k) \in \Theta(k)$ for $X = \mathcal{MR}, \mathcal{LMR}, \mathcal{SMR}$.*

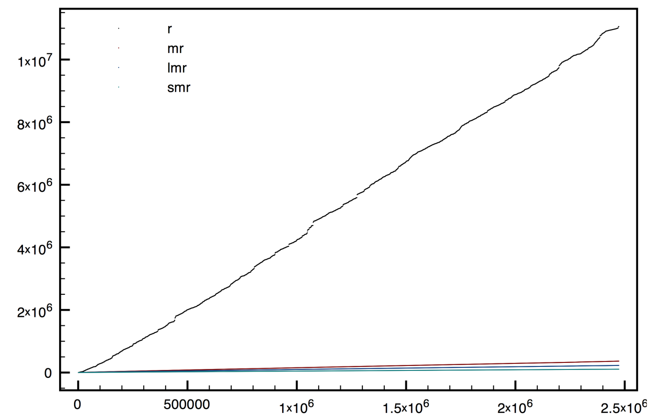
[1] This definition is due to Jacques Nicolas.



(a) bible



(b) E.coli



(c) world192

Figure 3.2: Number of all four classes of repeats for successive prefixes of `bible.txt` (a), `Ecoli` (b) and `world192.txt` (c)

X	$n_X(k)$	Proof	$Occs_X(k)$	Proof
$\mathcal{R}$	$\Theta(k^2)$	Lemma 2	$\Theta(k^2)$	Lemma 2
$\mathcal{MR}$	$\Theta(k)$	Lemma 1	$\Theta(k^2)$	Lemma 3
$\mathcal{LMR}$	$\Theta(k)$	Lemma 1	$\Omega(k^{\frac{3}{2}})$	Lemma 4
$\mathcal{SMR}$	$\Theta(k)$	Lemma 1	$\Theta(k)$	Lemma 5

Table 3.1: Upper and lower bounds for the number of normal, maximal, largest-maximal and super-maximal repeats; and for the total number of occurrences of these classes.

Proof. It is a known fact that the number of maximal repeats is $\mathcal{O}(n)$. The upper bound is therefore trivial

For the lower bound, we will prove that it holds for super-maximal repeat, and therefore also for maximal and largest-maximal. Consider the family of sequences $s_k = a_1a_2\ldots a_k|a_1a_2|a_2a_3|\ldots|a_{k-1}a_k$, over an alphabet of size $2k$ ($\Sigma(s_k) = a_1, \ldots, a_k, |_1, \ldots, |_{k-1}$). Note that $n = |s_k| = 3*(k-1)+k = 4*k-1$. There is no repeat of size bigger than two. Moreover, every pair a_ia_{i+1} is a super-maximal repeat: every such pair appears two times and because they are all different and there are no longer repeats, no other repeat is a superstring of them. So there are $k-1$ such repeats, which gives the lower bound $\Omega(k)$ for the number of super-maximal repeats. □

Lemma 2. $Occs_{\mathcal{R}}(k) \in \Theta(k^2)$

Proof. The upper bound follows trivially from the number of possible substrings.

For the lower bound, consider $s_k = a_1\ldots a_ka_1\ldots a_k$ of size $2k$ over an alphabet of size k. Every substring of $a_1\ldots a_k$ is a repeat and they are $\mathcal{O}(k^2)$ such substrings. □

Lemma 3. $Occs_{\mathcal{MR}}(k) \in \Theta(k^2)$

Proof. Again, the upper bound follows trivially from the number of possible substrings.

For the lower bound consider $s_k = ax^kb$ of size $k+2$. Every x^i is a repeat and each such repeat has an occurrence with left context a and right context x and another occurrence with left context x and right context b. Thus, they are maximal. x^i appears $k-i+1$ times and $\sum_{i=1}^{k-1} k-i+1 \in \mathcal{O}(k^2)$. □

Lemma 4. $Occs_{\mathcal{LMR}}(k) \in \Omega(k^{\frac{3}{2}})$.

Proof. Consider the family of sequences $s_k = x|xx|xxx|\ldots|x^k$, over an alphabet of size k. The size n of s_k is $k + \sum_{i=1}^{k} i = k + \frac{k*(k+1)}{2}$.

Every x^i for $i < k$ is a largest maximal repeat: it is repeated and there is one occurrence which no other repeat covers. This occurrence is always the first appearance of this repeat. Repeat x^{k-1} appears three times, x^{k-2} appears six times (its first occurrence thanks to which it is largest, two times in x^{k-1}, three times in x^k), and in general x^{k-i} appears $\sum_{j=1}^{i+1} j$. The total number of occurrences of all largest maximal repeats is then:

$$\sum_{i=1}^{k-1}\sum_{j=1}^{i+1} j = \sum_{i=2}^{k}\sum_{j=1}^{i} j = \sum_{i=2}^{k}\frac{i*(i+1)}{2} = \frac{1}{2}\sum_{i=1}^{k} i^2 + \mathcal{O}(k^2) = \mathcal{O}(k^3)$$

As $|s_k| \in \mathcal{O}(k^2)$, this gives the claimed lower bound. □

Lemma 5. $Occs_{\mathcal{SMR}}(k) \in \Theta(k)$

Proof. The lower bound is a direct corollary from Lemma 1 which bounds the number of different super-maximal repeats.

For the upper bound, note that no occurrence can completely be contained in another occurrence. Therefore, each occurrence has at least one position that makes it super-maximal and there exists then an injective function from the set of occurrences to the set of positions.

□

In Fig. 3.2 we computed the number of these repeats on successive prefixes of the Large corpus. A clear difference emerges between the DNA sequence and the others: not only is the absolute number of the different number of repeats higher, but one can see an escalated behavior of the normal repeats when entering a repeat-rich zone. The other type of repeats however, seem to behave more uniformly. Finally, an expected difference can be appreciated between the semi-structured text (`world192.txt`) and the natural language one (`bible.txt`): the total number of repeat of the former is almost doubled but the difference between the higher classes of repeat is less noticeable.

3.2.2 Computation

An *ESA* enables computation in $\mathcal{O}(n)$ of maximal repeats [110, 134] and super-maximal repeats [2, 110]. Gusfield also states that near-supermaximal repeats can be computed in linear time using a suffix tree, though without giving much detail.

We present here a linear algorithm that computes *directly* largest-maximal repeats using an ESA. Our emphasis on "directly" is because an easier linear algorithm to compute them consist in filtering maximal repeats. This algorithm is due to Jacques Nicolas and we outline it here shortly. We start marking all maximal repeat occurrences that are not covered to the right by another repeat. These correspond to internal nodes with at least one leaf-daughter in suffix trees and suffix array indexes i such that $lcp[i] = |w|$ with w the maximal repeat. There are only a linear number of such occurrences. Like usual with suffix data structures, the treatment for the right context — which normally is straightforward — differs from the one for the left context. To see that there is no other repeat that covers each one of them on the left, we perform the following algorithm: we save all these positions in an array of size n. Then, we read this array from left to right, computing for each occurrence o of maximal repeat w its end position ($o + |w|$). If another occurrence $o' > o$ of $w' \neq w$ is contained in this coverage ($|w'| + o' < |w| + o$), then we eliminated occurrence o'. At the end, only occurrences of largest maximal repeats remain.

Our algorithm avoids to compute all maximal repeats, thus reducing memory storage. As we have already seen, a not-so-easy-computable characteristic for an occurrence is to be *left-context unique*. This is that the symbol to the left of this occurrence differs from all the left context of all others occurrences. On a suffix-tree, the left context of leaf v (which we denote by $lc(v)$) is the $(i-1)$-th symbol of the sequence, where i is the position defined by leaf v.

We traverse the lcp-interval tree [2] and for each index we calculate if this leaf is left-context unique. If it is, we mark the corresponding father node as a largest-maximal repeat. We based the notation of the algorithm on the one of Puglisi et al. [193] to calculate maximal repeats.

A standard way of reasoning with repeats over the suffix array is to regard the decreasing or increasing of the values in the *lcp* array. A decreasing value correspond to the end of one or more trees. In this case, the leaf correspond to the first of those trees that ends. If the *lcp* value remains equals, then the leaf is the one of the current tree. Finally, if it is increasing, then a new tree started, and the leaf corresponds to this new one. Pseudo-code for our algorithm is given in Algorithm 2.

Algorithm 2 Calculation of largest-maximal repeats with an enhanced suffix array

LMR-SuffixArray(lcp[ ])

Input: the LCP array

Output: all the largest maximal repeat in the form $\langle length : start, end\rangle$ where *start* and *end* defines the interval of occurrences over the suffix array

1: *lb, lcp, islmr = 0,0,false*
2: $stack.push(\langle lb, lcp, islmr\rangle)$
3: **for** i from 1 to n **do**
4:   lb $\hookleftarrow$i
5:   **if** $lcp[i+1] < stack.top().lcp$ **then**
6:     $stack.top().islmr \hookleftarrow stack.top().islmr \vee isLCUnique(i)$
7:     **while** $lcp[i+1] < stack.top().lcp$ **do**
8:       $lb \hookleftarrow stack.top().lb$
9:       $\langle p, lcp, islmr\rangle \hookleftarrow stack.pop()$
10:       **if** $islmr$ **then**
11:         **print** $\langle lcp : p, i\rangle$
12:       **end if**
13:     **end while**
14:   **end if**
15:   **if** $lcp[i+1] > stack.top().lcp$ **then**
16:     $stack.push(\langle lb, lcp[i+1], false\rangle)$
17:   **end if**
18:   **if** $i = lb$ **then**
19:     $stack.top().islmr \hookleftarrow stack.top().islmr \vee isLCUnique(i)$
20:   **end if**
21: **end for**

The test $i = lb$ (line 18) takes care of the case where the current leaf was already tested because it belongs to a tree that was *popped* from the stack

Because the number of times the while loop is executed is linear (one for each right largest-maximal repeat), the execution time is $\Theta(n)$, supposing that $isLCUnique$ is constant. This function returns true if $lc(i)$ does not occur as left

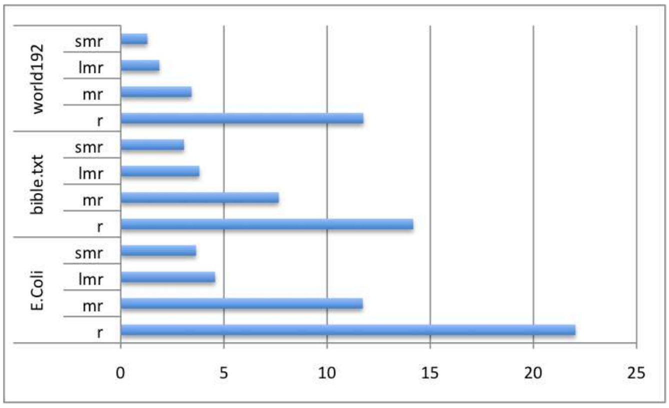


Figure 3.3: Execution time (in seconds) of four different classes of repeat for the Large corpus. Time is given in seconds, represents user time on a Intel(R) Xeon(R) 2.93GHz CPU with 6GB of memory running Fedora 12 (Constantine) and is averaged over 20 executions. See the text for details on the algorithms used.

context of any other leaf of the immediate super-tree of i. To achieve constant computation of this function, we extend the idea used by Puglisi et al. [193] to obtain a linear time for the computation of supermaximal repeats. Auxiliary arrays *next* and *prev* are used that keep the next (previous) occurrence of $lc(i)$ as a left context. That is, for a position i, $prev[i] = j$ ($next[i] = j$) if $lc(j) = lc(i)$ and for all k such that $j < k < i$ ($j > k > i$), $lc(k) \neq lc(i)$. This computation can be done in linear time (linear in $\max(|\Sigma|, n)$). Finally, for each tree we need to know where it does finish. We store this information in another array (*endTree*) where in each position the end of the current tree of this position is stored. This also can be computed in linear time, by a previous traversal of the *lcp-interval* tree. Finally, *isLCUnique* correspond to $prev[i] < stack.top().lb \wedge next[i] > endTree[i]$.

Algorithm 2 permits also to calculate in linear time what Gusfields [110] called the *degree* of super-maximality of a largest-maximal repeat: the percentage of occurrences that are left-context unique.

We implemented algorithms to recover all four classes of repeats, based on an enhanced suffix array. For the maximal repeat, we used the algorithm from Abouelhoda et al. [2], Puglisi et al. [193], for super-maximal repeats again [193], Algorithm 2 for largest-maximal repeat and the trivial one[2] for computing normal repeats. In Fig. 3.3 we compare the execution time of these algorithms on the Large corpus. We notice a huge difference between the time required to compute simple repeats and the time required for all others repeat. A second difference is between maximal repeats and the two other classes, whose time is comparable (though super-maximal repeats require consistently less in all three cases).

[2]Every time the *lcp* value increases, add a repeat to the stack, and output it if the *lcp* values decreases

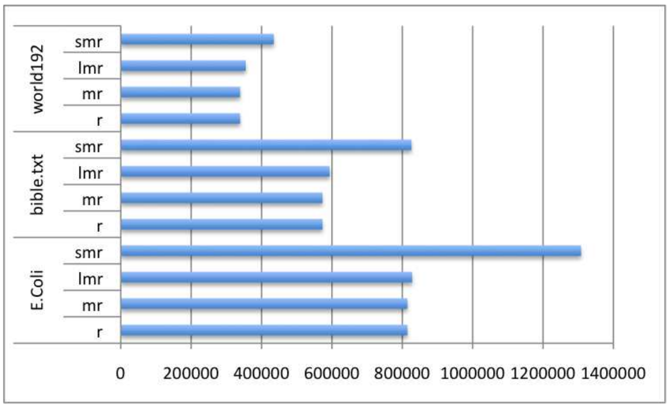


Figure 3.4: Size of the final grammar obtained with IRR-MC of the four different classes of repeats for the Large Corpus.

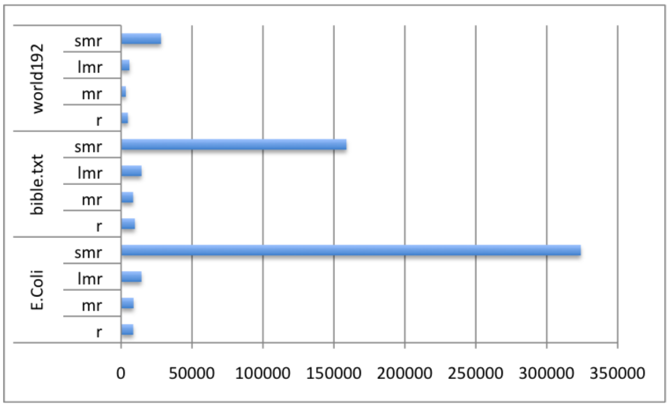
3.2.3 Use in IRR

In the IRR schema however, the computation of the repeats has to be done in every iteration, so the results from Fig. 3.3 cannot be extrapolated to the total execution time of IRR. Here we analyse briefly this execution time, and compare the size of the grammars obtained at the end. We focus only on IRR-MC, the algorithm we identified as generating the smallest grammars (see Sect. 2.7.2) . Instead of searching over all repeats, we replaced $\hat{\mathcal{R}}(s)$ in Algorithm 1 (p. 34) with the other classes of repeats. On Fig. 3.4 we report the length of the final grammars obtained with IRR-MC. Except for `E.coli`, the final grammar obtained with normal repeats and with maximal repeats are exactly the same. The final size of the grammars obtained with largest-maximal repeats are slightly bigger while the difference of the grammars obtained with super-maximal repeats is considerable.

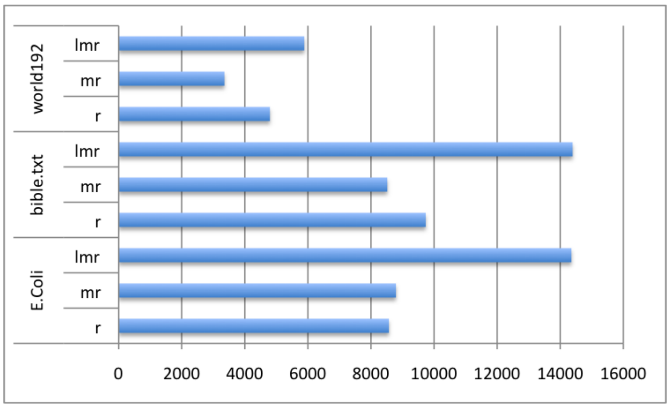
Regarding the gain in execution time, using super-maximal repeats not only results in bigger grammars, but also takes much more time to compute as can be appreciated in Fig. 3.5(a). This is probably due to the much higher number of iterations that are executed because of the use of large repeats. The difference between the use of largest-maximal repeats and maximal repeats is less than the difference in the one-time execution that we measured in Fig. 3.3 (compare with Fig. 3.5(a)). This holds also with respect to simple repeats: we suppose that in the last iterations, the difference between these three classes of repeats become less noticeable.

Of course, the improvement considering only maximal repeat varies depending of the sequence. On the 557 Knt (kilo-nucleotides) sequence of the maize (*zhea mays*) mitochondrion, known for having a large number of repeats, we reached a speed-up of 6.6 times compared to the use of IRR-MC using normal repeats.

Comparing the speed-up with the final grammar size, we choose to focus on the use of maximal repeats instead of considering all simple repeats. For the IRR algorithms, this produces little change. For IRR-ML, the chosen word is always a maximal repeat and for IRR-MF, there is always a maximal repeat



(a)



(b)

Figure 3.5: User time (in seconds, averaged over 10 executions) for computation of IRR-MC considering four different classes of repeat for the Large Corpus. Run on a Intel(R) Xeon(R) 2.80GHz CPU with 32GB of memory running Red Hat 4.1. Fig. (b) is a detail of Fig. (a).

that has maximal score[3]:

Proposition 3.

1. *If* $f_{ML}(\omega, G) = \max_{\alpha \in \mathcal{R}(r(G))} f_{ML}(\alpha, G)$ *then* ω *is a maximal repeat.*

2. *There is always a maximal repeat* ω *s.t.* $f_{MF}(\omega, G) = \max_{\alpha \in \mathcal{R}(r(G))} f_{MF}(\alpha, G)$

For the case of IRR-MC, we do not have an equivalent property. It could happen that a repeat that maximises f_{MC} is not maximal. Consider for instance the case of a non-maximal repeat ω with two occurrences, both of them with context $\langle a, a\rangle$. If both occurrences of $a\omega a$ overlap (because they occur at position i and $i + |\omega| + 1$ for some i), then $a\omega a$ would not be considered and the best repeat becomes ω. Here we will characterise the condition when a non-maximal repeat maximizes f_{MC}.

If ω is a repeat, then there is exactly one maximal repeat that contains ω and appears the same number of times. We call this maximal repeat $mr(\omega)$. We are interested in non-maximal repeats ω such that $f_{MC}(\omega, \mathcal{P}) > f_{MC}(mr(\omega), \mathcal{P})$. Note that $|pos_{r(G)}(\omega)| = |pos_{r(G)}(mr(\omega))| + k_1$ and $|mr(\omega)| = |\omega| + k_2$ for some positive k_1, k_2, this is, $mr(\omega)$ is k_2 symbols longer the ω and have k_1 occurrences that must be eliminated to have a maximal non-overlapping list. Replacing in the definition of f_{MC}:

$$\begin{array}{lccc} & (o_{\mathcal{P}}(\omega) - 1) * (|\omega| - 1) & > & (o_{\mathcal{P}}(\omega) - k_1 - 1) * (|\omega| + k_2 - 1) \\ \equiv & \frac{k_1}{k_2} & > & \frac{o_{\mathcal{P}}(\omega) - 1}{|\omega| + k_2 - 1|} \end{array}$$

Supposing that $k_2 = 1$, this gives

$$|\omega| * k_1 > o_{\mathcal{P}}(\omega) - 1 \tag{3.1}$$

At the same time, supposing that the distribution over the sequence is i.i.d., the probability that a word w is a non-maximal repeat is $2 * \left(\frac{1}{|\Sigma \cup \mathcal{N}|}\right)^{(o_{\mathcal{P}}(w) - 1)}$ (it must have all its left-context equal, and all its right-context equal). Let us remind that $|\mathcal{N}|$ increases by one in each iteration of IRR. Both equations indicate that in order to find a case where f_{MC} is maximal for a non-maximal repeat, this repeat must have a low number of occurrences. However, in this case f_{MC} would assign it a lower score. So, in practice, such cases should not appear too frequently.

Our experiments confirmed this: in all instances but one of the DNA corpus, IRR-MC behaves as the version of IRR-MC that only looks at maximal repeats. In each iteration, both algorithms chose the same repeat and consequently at the end of the execution, both algorithms return the same grammar. File `vaccg`, where the two algorithms produce different grammars, presents an instance of the situation we described above, but the grammar returned by the algorithm that looks only at maximal repeat is only four symbols bigger than the one returned by IRR-MC. See Table 3.2.

On top of yielding almost equivalent results in faster time, the use of maximal repeats has the nice property that the grammar (a slightly modified version of) IRR returns are *irreducible*, independently of the score function being used.

[3]The original RePair (see Sect. 2.6.7) algorithm considers only digrams. In this case, a non-maximal repeat could be selected.

Theorem 3. *If IRR only considers maximal non-overlapping repeats (two non-overlapping occurrences must have different contexts), then the resulting grammar is irreducible.*

Proof. Recall the definition of an irreducible grammar (Def. 4, page 22). Condition 1 (the grammar is admissible) is trivially true for IRR algorithms, but Condition 4 (no repeat in the final grammar) may be violated by the IRR schema if it stops when no further improvement can be made. Nevertheless, it is enough to change the condition of the while loop in order to continue until G contains no repeats.

Condition 2 (all constituents are different) is harder to see. A clean demonstration is given in Charikar et al. [51, Lemma 6 and 7]. While their notion of *global algorithm* is different from IRR, the demonstration in these lemmas can be applied without modification to IRR.

Finally, an IRR algorithm may still violate Condition 3 (every non-terminal must appear more than once). Suppose for example that a non-maximal repeat α is chosen and replaced by N, and that every occurrence of α has as right context of a. If in a future iteration the repeat Na is chosen, then N would occur only once in the grammar. In Charikar et al. [51] a special kind of repeat is defined to avoid these cases. Instead of this, the use of maximal repeat gives a more general solution: if it is ensured that the selected word has at least two occurrences in his normalised list with different context, then the resulting grammar is irreducible.

□

So, any of the results of Kieffer and Yang [127] that applies for irreducible grammars applies to grammars obtained inside the IRR framework. This means in particular, that IRR grammars are universal codes.

3.3 Non-overlapping Occurrences

The total number of times a word occurs in a sequence can be easily computed in constant time using a suffix tree structure. But the exact computation of the number of non-overlapping occurrences ($|\mathcal{L}_{r(G)}(w)|$), is more complicated. The problem of computing this number is known as the **String Statistics Problem**. A solution is based on the construction of the **Minimal Augmented Suffix Tree** (MAST) [16] which permits to compute $|\mathcal{L}_{r(G)}(w)|$ in time $|w|$. The best known algorithm for the construction of a MAST is in $\mathcal{O}(n \log n)$ [38] and it builds in a first phase a suffix tree. So, even reducing the set of candidates to a linear number using maximal repeats, the total running time for a general IRR schema is still $\mathcal{O}(n^2 \log n)$ (the MAST must be created in every iteration), and requires the rather elaborate construction algorithm for a MAST.

We propose a much simpler approach: we ignore overlapping occurrences and instead of $|\mathcal{L}_{r(G)}(w)|$ we estimate it by the total number of occurrences of w in G ($|pos_{r(G)}(w)|$). While this score could be very different from the real contraction that could be achieved by replacing this repeat, our experiments (see Table 3.2) indicate, that over the DNA Corpus there is only a small difference between both grammars, and most of the times the version ignoring overlapping occurrences is actually smaller.

sequence	Size			Time		
	IRR-MC	Accel.	Δ	IRR-MC	Accel.	Δ
chmpxx	28,706	28,754	**-0.17**	20.61	10.02	**205**
chntxx	37,885	38,089	**-0.54**	33.92	16.8	**201**
hehcmv	53,696	53,545	**0.28**	65.48	32.21	**203**
humdyst	11,066	11,201	**-1.22**	3.99	1.73	**230**
humghcs	12,933	12,944	**-0.09**	49.34	5.5	**897**
humhbb	18,705	18,712	**-0.04**	19.62	5.01	**391**
humhdab	15,327	15,311	**0.1**	9.55	3.77	**253**
humprtb	14,890	14,907	**-0.12**	8.45	3.42	**247**
mpomtcg	44,178	44,178	**0.0**	55.44	24.6	**225**
mtpacga	24,555	24,604	**-0.2**	17.64	8.46	**208**
vaccg	43,701	43,491	**0.48**	54.95	23.12	**237**
average			**-0.13**			**299**

Table 3.2: Comparison between IRCC-MC and its accelerated version (using maximal repeats and not considering overlapping for score computation). Time is given in seconds and differences are given in percentage.

An advantage of only computing the non-overlapping occurrences list for the selected repeat is that the resulting IRR schema, using maximal repeats, decreases the complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, for any score whose computation time is constant. This requires only standard techniques (computation of maximal repeats). Special care should be taken that the chosen repeat do have more than one non-overlapping occurrence, in which case adding this production rule would actually increase grammar size. In such cases we take the next best maximal repeat.

Combining the improvements from this section (ignore overlapping occurrences) and the one of Sect. 3.2 (use of maximal repeats instead of normal) gives an accelerated version of IRR-MC. In Table 3.2 we indicate the time that it took IRR-MC to run on each of the sequence of the DNA Corpus, and the ratio of the accelerated version and the original. Speed-up varies from two (`chntxx`) to nine (`humghcs`). Except otherwise stated, from now on we will suppose both of these improvements are included in the algorithm.

3.4 In-place Update of Suffix Array

3.4.1 Motivation

In this section we propose an algorithm to efficiently update a suffix array, after substituting a word by a new character in the indexed text. This is the main task in the IRR schema, and differently from what we saw before, we are not concerned with the detection of an interesting word, but with maintaining the associated index structure that will permit to find the words in the next iteration.

Efficient implementation of an elaborate choice of repeat often requires the use of data structures from the suffix-tree family. These index structures are well suited for efficient computations on repeats but they have to be built at initialisation, and then updated at each step of the algorithm with respect to sequence modifications. Yet, as pointed out by Apostolico and Lonardi [13], most of the published work on dynamic indexing problem [201], by updating a suffix tree [48, 88, 90, 109, 162] or a suffix array [208], focuses on localised modifications of the string. They do not seem appropriate for efficiently replacing *more than one* occurrence of a given substring, as they would require one update operation for each occurrence.

Thus, index structures have usually to be built from scratch at each step of the algorithm. To our knowledge, only some implementations of LongestFirst [138, 168], updates a suffix tree data structure after the deletion of all occurrences of a word. However, their updating scheme are specific to the longest matching substrings and seems difficult to adapt to other strategies.

We propose here a solution to the problem of updating efficiently an index structure while replacing some non-overlapping occurrences of a word of the indexed text by a new symbol. The first originality of our approach relies on the use of enhanced suffix arrays instead of suffix trees. A simple way of updating suffix array (instead of enhanced suffix array, thus without the same efficiency objective) by lazy bubble sort has been used in Nevill-Manning and Witten [177]. We propose here, to take advantage of the internal order offered by enhanced suffix arrays, to simultaneously handle groups of indices.

3.4.2 Double-linked Enhanced Suffix Array

The algorithm presented below consists mostly of moving and deleting lines of the *ESA* and keeping *lcp* consistent. In order to avoid shifting sets of indices, we link consecutive indices using two additional arrays called *next* and *prev*. Thus, $next[i]$ (resp. $prev[i]$) gives the index of the next (resp. previous) valid entry in the *ESA*. Initially, $next[i] = i + 1$ and $prev[i+1] = i$. So, if for example the index i must be deleted, that can be easily done by setting $next[prev[i]]$ to $next[i]$ and $prev[next[i]]$ to $prev[i]$. We call the set *ESA* plus *next* and *prev* arrays the ESA_{DL} for **Double-Linked Enhanced Suffix Array**.

It is worth noticing that an ESA_{DL} does not have the exact same properties as an *ESA*. Indeed, going from an index i to index $i+j$ may be done in constant time on an *ESA*, while this operation in an ESA_{DL} requires $\mathcal{O}(j)$ time, as the *next* array has to be used j times. Moreover, because of ESA_{DL} lines moving, the result of indices comparison may not coincide with the order of the associated suffixes. For instance, index i may correspond to a suffix with a lexicographically order greater than a suffix corresponding to index j, even if $i < j$. Anyway, an ESA_{DL} still allows the detection of repeats (general repeats, maximal, largest-maximal or super-maximal repeats) in linear time, because the involved algorithms advance one by one over the arrays like most of the algorithm over *ESA* (a notable exception is the algorithm searching for a substring proposed in Sim [222]). Finally, we remark that the standard *ESA* can directly be recovered in one simple pass from ESA_{DL}.

We propose an *in-place* solution, where we always work with the same arrays and only update the values of their fields. Moreover, during the whole process, we modify only the *prev*, *next* and *lcp* arrays. Arrays *sa* and *isa* remain unchanged. This approach forces to extend the in-place behavior to the sequence: we also add two arrays to imitate a double-linked list over the sequence: the j^{th} position after position i, is denoted by $i \oplus j$. We compute $i \oplus j$ using links between sequence positions, indicating for each position its successor. Similarly

$i \ominus j$ points to the j^{th} position before i. We define that, if $i \oplus j$ (respectively $i \ominus j$) is out of range, then $i \oplus j = n + 1$ (respectively $i \ominus j = -1$).

As we have seen, an IRR-like algorithm proceeds by steps. At each step, the alphabet grows because of the introduction of a new character: Σ_k will denote the alphabet at step k. At each step k the algorithm **i)** finds a repeat w_k in a sequence $\tilde{s}^{(k)}$ defined on the alphabet Σ_k and returns a list o_k of non-overlapping occurrences of w_k **ii)** updates the sequence $\tilde{s}^{(k)}$ and its associated ESA_{DL} replacing the given occurrences of w_k by a single new character c_k, thus defining a new alphabet $\Sigma_{k+1} = \Sigma_k \cup \{c_k\}$. The modified sequence is then called $\tilde{s}^{(k+1)}$. Until now, we supposed $o_k = pos(w_k)$ or $o_k = \mathcal{L}(w_k)$ but the solution we present here works for any non-overlapping list of occurrences. The whole iterative process stops either if no more repeats are found in the sequence or after a fixed number of iterations.

Our contribution focuses on updating the ESA_{DL}, at each step k of this algorithm (part **ii**).

In the next sections, we describe how to perform the three tasks needed for updating an ESA_{DL} at each step k: **1)** delete indices of suffixes starting inside a w_k occurrence; **2)** move indices with respect to the alphabetic order of c_k; and **3)** update *lcp* array with respect to recoded occurrences of w_k by one single character. Note that a few values of the *lcp* array are also modified during step 1 and 2, but only as a consequence of deletions and moves.

Note the difference with the IRR algorithm (Algorithm 1 (p. 34)) that the w_k is only replaced inside the original sequence, not in the right-hand side of previous introduce rules. To adapt the algorithm here to this general case, a fourth step would be necessary, that *inserts* in the enhanced suffix array the just replaced word.

To better understand the different steps of the algorithms and the modifications they perform over the suffix array, we will define the concept of **left-context tree**. It is worth noticing that we present this structure in order to help the understanding of our approach and that it is not actually implemented.

The left-context tree

One of the most useful characteristics of a suffix array is that all indices corresponding to suffixes starting with the same word (substring) correspond to an adjacent block. We define here the corresponding concept of word interval. Based on this, we will define the *left context tree* of a word ω where the nodes correspond to a left-context of ω.

An ω-interval is the set $\{k : \exists \ell,\ k = isa[\ell] \wedge \tilde{s}[\ell..\ell + |\omega| - 1] = \omega\}$. This can also be denoted as an $[i..j]$-interval, where i and j are respectively the lowest and highest indices of an ω-interval. Let us note that different words can share the same interval. More precisely, any pair of words ω and $\omega\alpha$ share the same interval if each occurrence of ω is followed by α.

This definition is thus slightly more general than the definition of ω-interval given by Abouelhoda, Kurtz and Ohlebusch [2], since in our approach ω-interval are defined also for words leading to implicit nodes of a compact suffix tree, and not only to internal nodes.

The *left-context tree of* ω ($\omega \in \Sigma^*$) for a sequence $\tilde{s}$ is an implicit tree whose nodes are v-intervals ($v \in \Sigma^*$) such that:

- the root is the ω-interval
- for each v-interval node corresponding to a non-empty interval, its children are all the av-intervals, for all $a \in \Sigma$
- the leaves are empty intervals

Given the *isa* array, it is easy to obtain the parent of a node. Let $[i..j]$ be an av-interval node. Given $k \in [i..j]$, $isa[sa[k]+1]$ is an index belonging to the v-interval. Inversely, $isa[sa[k]-1]$ belongs to one of the child interval. The exact child depends on the symbol at $\tilde{s}[sa[k]-1]$. We introduce the *successor* and *predecessor* notations:

$$successor(i) = \begin{cases} isa[sa[i] \oplus 1] & \text{if } sa[i] \oplus 1 \neq n+1 \\ n+1 & \text{otherwise,} \end{cases}$$

and

$$predecessor(i) = \begin{cases} isa[sa[i] \ominus 1] & \text{if } sa[i] \neq 0 \\ -1 & \text{otherwise.} \end{cases}$$

One may remark that *predecessor* is the equivalent of the "*suffix link*" in a suffix tree [233].

The problem that an *ESA* update algorithm must face is that the changes over the occurrences of a word ω not only affect the ω-interval, but also some of the $v\omega$-intervals ($v \in \Sigma^*$). The core of our algorithm is based on moving a $v\omega$-interval in constant time, using the two following properties implied by the internal order of suffix arrays:

Proposition 4. *Let $[i..j]$ be an v-interval ($v \in \Sigma^*$), and $k_1, k_2 \in [i..j]$ with $k_1 > k_2$ and such that $predecessor(k_1)$ and $predecessor(k_2)$ belong to the same av-interval ($a \in \Sigma$). Then $predecessor(k_1) > predecessor(k_2)$.*

Proposition 5. *With $i < j$, the longest common prefix between $\tilde{s}[sa[i]..]$ and $\tilde{s}[sa[j]..]$ is $\min\limits_{k \in [next[i],j]} lcp[k]$.*

3.4.3 Algorithm

We now detail the three tasks for updating an ESA_{DL} while replacing a set of occurrences o_k of a word w_k by a simple character c_k.

Delete indices of suffixes occurring inside w_k substituted occurrences

By replacing the word w_k by a single letter, the sequence is compressed and so is its ESA_{DL}: consequently, any suffix of sequence $\tilde{s}^{(k)}$ starting inside an w_k substituted occurrence must be deleted. Thus for i in o_k and for ℓ in $[1, |w_k|-1]$, suffix $\tilde{s}^{(k)}[i \oplus \ell..]$ and the associated index in the suffix array $j = isa[i \oplus \ell]$ have to be removed. We simulated this deletion by *jumping over it* by setting *next* and *prev* arrays to their previous and next index: $next[prev[j]] \hookleftarrow next[j]$ and $prev[next[j]] \hookleftarrow prev[j]$. Furthermore, the *lcp* value of the index following j ($lcp[next[j]]$) has to be modified according to the deletion of index j. As a consequence of Proposition 5, after the deletion of index j, the longest common prefix of index $next[j]$ is equal to the minimal longest common prefix value of indices j and $next[j]$.

An example is shown in Fig. 3.6 where the deletion of index j affects $lcp[next[j]]$ that now should contain the length of the longest common prefix between $ATGT$

index	*prev*	*next*	*lcp*	suffix
$j-1$	$j-2$	$\cancel{j}\ j+1$	4	$ATAC\ldots$
$\cancel{j}$	$\cancel{j-1}$	$\cancel{j+1}$	$\cancel{2}$	$\cancel{ATGA\ldots}$
$j+1$	$\cancel{j}\ j-1$	$j+2$	$\cancel{3}\ \mathbf{2}$	$ATGT\ldots$

Figure 3.6: Deletion of index j

and $ATAC$ which is 2, equal to the longest common prefix of $ATGT$, $ATGA$ and $ATAC$.

Algorithm 3 presents the procedure for deleting indices. The notation END refers to the last index of the suffix array ($prev[n+1]$).

Algorithm 3]

$delete\_indices(ESA_{DL}^{(k)}, w_k,\ o_k)$ Delete indices at step k, replacing w_k by c_k

```
for i ∈ o_k do
  for ℓ ∈ [1, |w_k| − 1] do
    j ↤ isa[i ⊕ ℓ]
    if next[j] ≠ END then
      lcp[next[j]] ↤ min(lcp[j], lcp[next[j]])
    end if
    next[prev[j]] = next[j]
    prev[next[j]] = prev[j]
  end for
end for
```

Move indices, with respect to the alphabetic order of c_k

After replacing the word w_k by the new character c_k, some ESA_{DL} lines may be misplaced with respect to the chosen order of c_k in Σ_{k+1}. Indices in the w_k-interval are potentially misplaced. In fact, for $v \in \Sigma_k^*$, indices inside an vw_k-interval are misplaced if the substitution of w_k into c_k affects their lexicographical order with respect to the previous and next index over the suffix array. Thus, lines belonging to node-intervals of the left-context tree of w_k may have to be moved.

In our approach, we decided to give to c_k the largest rank in the lexicographic order of the alphabet Σ_k, *i.e.* $\forall\, a \in \Sigma_k : a \prec c_k$.

With respect to this arbitrary choice, the w_k-interval is moved to the end of the suffix array. Furthermore, for any $v \in \Sigma_k^*$, the vw_k-interval is moved after the last index of the v-interval.

If a vw_k-interval is already at the end of the v-interval (it is already well ordered), for any $v' \in \Sigma_k^*$, the $v'vw_k$-interval is also at the end of the $v'vw_k$-interval and does not have to be moved.

Based on this property, our algorithm uses a recursive approach in order to move groups. The recursion starts on the initial w_k-interval. During recursion, if the group of a vw_k-interval is moved, the recursion continues on groups of avw_k-intervals, with $a \in \Sigma_k$.

l	sa	lcp	suffix
0	8	0	\$
~~1~~	~~1~~	~~0~~	~~$AAGAA\ldots$~~
~~2~~	~~4~~	~~3~~	~~$AAGC\$$~~
3	2	~~0~~1	$A\boxed{GA}AG\ldots$
4	5	~~2~~0	$AGC\$$
5	7	0	$C\$$
6	0	~~0~~0	$\boxed{GA}A\boxed{GA}\ldots$
7	3	4	$\boxed{GA}AGC\ldots$
8	6	~~1~~0	$GC\$$

Figure 3.7: Moves induced by substituting GA by c_1. $lcp[3]$ was 0 after the delete step and $lcp[7]$ will be updated during the third step.

From a theoretical point of view, the algorithm starts on the root of the left-context tree of w_k and if the group corresponding to the interval of the node is moved, it recursively treats its children in a breadth first traversal (a FIFO is used).

In practice, the recursion on a vw_k-interval works as follows:

1. detects the end position of the vw_k-interval,
2. detects the end position of the v-interval,
3. if necessary:
   - 3.a. moves the group to the end position of the v-interval,
   - 3.b. calls the recursion on predecessors of indices of the group.

During a call on *predecessor* of an index of the group, either this is the first time the matched group is called and by construction the call is done on its first element, or the group was already treated, and the recursion stops.

The algorithm for this step is shown in Algorithm 4. This recursive function receives three parameters besides the data structures: the starting position of the group, the current depth over the left-context and a boolean flag (see below).

In first place, the end of the vw_k-interval is found (lines 5, 6 and 8).

This is done from the first element of the interval, following the *next* array while the visited index corresponds to a suffix starting with vw_k ($lcp[i] \geq |v| + |w_k|$). After finding the extremes of the group, the destination index of this group according to the chosen order for the new character is found (lines 11, 12 and 15). This is done by finding the end of the v-interval in the same way ($lcp[i] \geq |v|$).

Moving the group to its new position is now simple and is done in constant time. Thanks to the well-ordered property of the suffix array, the whole interval is moved by changing only the delimiting positions. Let $i_{start}, i_{end}, i_{dest}$ be respectively the starting and ending positions of the vw_k-interval, and the last position of the v-interval. Moving the group $[i_{start}, i_{end}]$ to the position after

Algorithm 4 In-place update of a suffix array: update order]Restore consistency of suffix array order

$update\_order(ESA^{(k)}_{DL}, w_k, o_k, i_{start}, depth, move)$
  if Couple $(i_{start}, depth)$ already treated during another recursion call **then**
    End procedure
  end if
  $i \hookleftarrow i_{start}$
  while $i \neq END \wedge lcp[next[i]] \geq depth + |w_k|$ **do**
    $i \hookleftarrow next[i]$
  end while
  $i_{end} \hookleftarrow i$
  $minLCP \hookleftarrow min_{j \in [i_{start}, i_{end}]} lcp[j]$
  if $move$ **then**
    while $i \neq END \wedge lcp[next[i]] \geq depth$ **do**
      $i \hookleftarrow next[i]$
    end while
  end if
  $i_{dest} \hookleftarrow i$
  if $i_{end} \neq i_{dest}$ **then**
    $lcp[next[i_{end}]] \hookleftarrow min(lcp[next[i_{end}]], minLCP)$
    $lcp[i_{start}] \hookleftarrow depth$
    if $i_{start} = i_{first} \wedge depth \neq 0$ **then**
      $i_{first} \hookleftarrow next[i_{end}]$
    end if
    move_group$(i_{start}, i_{end}, i_{dest})$
  else
    $lcp[i_{start}] \hookleftarrow min(lcp[i_{start}, depth)$
    $move \hookleftarrow false$
  end if
  $i \hookleftarrow i_{start}$
  while $i \neq next[i_{end}]$ **do**
    $newdepth \hookleftarrow depth +$ (**if** $predecessor(i) \in o_k$ **then** len **else** 1)
    if $move \vee (sa[prev[predecessor(i)]] > newdepth \wedge sa[prev[predecessor(i)]] \oplus newdepth \in o_k)$ **then**
      $update\_order(ESA^{(k)}_{DL}, w_k, o_k, predecessor(i), newdepth, i_{dest} \neq i_{end})$
    end if
    $i \hookleftarrow next[i]$
  end while

i_{dest} is easily done by jumping over the group and *inserting* it into i_{dest} and $next[i_{dest}]$. See Algorithm 5 for details.

Two longest common prefix values are modified as a consequence of the deletion of the group and its insertion:

1. $lcp[next[i_{end}]]$: contains the value of the length of the longest common prefix between $prev[i_{start}]$ and $next[i_{end}]$, which according to Proposition 5, is the minimum of the *lcp* values of the group and itself

2. $lcp[i_{start}]$: we assign to it the value of *depth*, that is the correct value over $\tilde{s}_{k+1}$. This serves also to set a stop-point for future recursions calls (see below).

Algorithm 5 Move the group $[i_{start}, i_{end}]$ after the position i_{dest}

$move\_group(ESA_{DL}^{(k)}, w_k, o_k, i_{start}, i_{end}, i_{dest})$
  $next[prev[i_{start}]] = next[i_{end}]$
  $prev[next[i_{end}]] = prev[i_{start}]$
  $next[i_{end}] = next[i_{dest}]$
  $prev[next[i_{dest}]] = i_{end}$
  $next[i_{dest}] = start$
  $prev[i_{start}] = i_{dest}$

As i_{first} points to the first line over the suffix array that contains a selected repetition, we also update i_{first} (line 19) if this line is moved.

Fig. 3.7 shows the ESA_{DL} of sequence $GAAGAAGC$, where $w_1 = GA$ is substituted by c_1. One remarks that the initial interval of suffixes starting with GA (indices 6 and 7) is moved as well as suffix starting with AGA (index 3). Note also that suffix starting with $GAAGA$ has to be moved with respect to suffix $GAAGC$.

A special case

Once an interval is treated, the recursion continues either if the current group was moved, or in the special case described in what follows.

Consider for instance the following situation, where the substituted repeat is TA.

i	C**TA**TT**TA**C...
i+1	C**TA**TT**TA**G...
i+2	C**TA**<u>T**TA**...</u>,

and suppose that the TTA-interval containing the index $isa[sa[i+2] \oplus 3]$ (the underlined suffix in the figure) was already at its right position and therefore does not have to be moved. So, its children in the left-context tree are not considered for future moves, and as a consequence, neither is index $i+2$. Supposing that we cut the recursion here, that means that when treating the $CTATT$-interval, $lcp[i+2] = 5$. This interval ends at the index $i+1$, but because we use the lcp array to detect it, we also consider index $i+2$ as part of the $CTATT$-interval.

To resolve this special case, the recursion continues even when the current interval was not moved. In this case, it will never be necessary to move an interval, but maybe update some *lcp* values to set *stop-points* for future recursion calls.

This is the reason for introducing the last parameter in algorithm 4 (the boolean flag *move*). It differentiates the normal case (when it is necessary to detect the destination index and move the interval) from the case in which the current interval is considered only to set a *stop-point* at the first index of the interval. The recursion continues in both cases.

Filtering non substituted w_k occurrences Among each vw_k-interval, suffixes starting with vw_k where w_k is not substituted (whose position does not belong to o_k) may occur. The associated indices in the ESA_{DL} should not be moved with the vw_k-interval. Thus, before applying the recursive procedure previously exposed, a straightforward *filtering step* is applied. During the recursion, each line i of each group is first checked in order to detect if it corresponds to an index of a selected occurrence ($sa[i] \oplus depth \in o_k$). Once a non-selected occurrence is detected, we move it to the beginning of the group (before i_{start}). As previously mentioned, this also involves modifications of the *lcp* array for maintaining its consistency. This step is basically a simplified version of Algorithm 4. It adds an extra auxiliary array of size n to keep track, for each index, of the last depth with which it was analysed.

Update lcp values after the substitution of w_k occurrences to a single character

The substitution of any occurrence of w_k of length $|w_k| \geq 2$ by c_k of length 1 involves the modification of the length of all common prefixes involving such an occurrence.

In the previous step, it was trivial to update the *lcp* values of the border lines. However, in this step, we update the *lcp* values of the internal position of the intervals. Straightforwardly subtracting $|w_k| - 1$ from each internal *lcp* value misses the cases where the common prefix between two successive suffixes include more than one occurrence of w_k, or even worse, a part of a occurrence (consider for instance the example shown in Sect. 3.4.3).

So we traverse again the left-context tree of w_k. Contrary to the moving step, where it was possible to move one line several times, in this step we update each *lcp* index only once. To do this, we recalculate all the *lcp* values for the root (w_k-interval) and use this information to update the *lcp* of the other intervals.

As a consequence of Propositions 4 and 5, the *lcp* between two indices of the same interval-node is simply one plus the *lcp* between their successor indices belonging to the parent interval-node:

Let i, j belong to the same aw-interval and let us assume that $i > j$.

Then $lcp(\tilde{s}[sa[i]..], \tilde{s}[sa[j]..]) = \min_{\ell \in [next[successor(i)], successor(j)]} lcp[\ell]$

With this inductive approach, it is sufficient to re-calculate the *lcp* of only the first interval (the root of the left-context tree) as shown in Algorithm 6.

During the iterative call, if an index that is already treated appears, it is skipped. Indeed, its *lcp* value is then up-to-date. The pseudo-code for this step is exposed in Algorithm 7.

Because in each iteration we use the value of all the lines of the previous group, we traverse once again the left-context tree in a breadth-first order.

Algorithm 6 Calculate the value of the *lcp* for index i

$recalculate\_lcp(ESA_{DL}, i)$
  $lcp[i] \hookleftarrow 0$
  if $prev[i] \geq 0$ **then**
    $i \hookleftarrow sa[i]$
    $j \hookleftarrow sa[prev[i]]$
    while $i < n \wedge j < n \wedge s[i] = s[j]$ **do**
      $i \hookleftarrow i \oplus 1$
      $j \hookleftarrow j \oplus 1$
      $lcp[i] \hookleftarrow lcp[i] + 1$
    end while
  end if

Algorithm 7 Update *lcp* of step k

$update\_lcp(ESA^{(k)}_{DL}, w_k, o_k)$
  $q \hookleftarrow queue()$
  for $i \in o_k$ **do**
    $recalculate\_lcp(ESA^{(k)}_{DL}, isa[i])$
    $q.push((predecessor(isa[i]), 1))$
  end for
  while not *q.empty()* **do**
    $(i, depth) \hookleftarrow q.top$
    $q.pop$
    if $i \geq 0 \wedge lcp[i]$ not already updated $\wedge\, lcp[i] \geq depth$ **then**
      $lcp[i] \hookleftarrow \left(\min_{j \in [next[successor(prev[i])], successor(i)]} lcp[j] \right) + 1$
      $q.push((predecessor(i), depth + 1))$
    end if
  end while

3.4.4 Efficiency

Time efficiency

The worst case time complexity of the update algorithm is loosely bounded by $\mathcal{O}(n^2)$. A better bound on time complexity could be obtained by considering amortised complexity over the overall IRR schema, but it will still be unlikely to be better than the $\mathcal{O}(n)$ complexity required for building the suffix array from scratch. Nevertheless, the algorithms building suffix arrays that currently perform best in practical cases, are not the linear ones (see Schürmann and Stoye [214] for a description of the different suffix array construction algorithms and their strengths). We propose in this section to evaluate the practical efficiency of our update algorithm, comparing it to the standard approach that builds the suffix array from scratch

A prototype implementing the proposed algorithm has been developed using the *C++* language[4]. It has been tested on different types of text. For the sake of brevity, here we only report the results on the Canterbury and Large Corpus (see Sect. A). Results on other corpora can be found on our internet site.

To compare the execution time with a building from scratch approach, we used three different suffix array creation algorithms: the linear time one proposed Kärkkäinen and Sanders [123], the non-linear algorithm of Larsson and Sadakane [140] and the Induced Sorting algorithm of Zhang et al. [247] (again a linear one). The source code of the first two were retrieved from the web sites specified in the associated articles. Note that Kärkkäinen and Sanders' code "strives for conciseness rather than for speed" [123]. For the Induced Sorting algorithm, we used the optimised implementation of Mori [166].

In the last years, suffix array creation algorithms has proven to be a rich field of research. New strategies and improvements are proposed each year, and for a complete taxonomy of the state of art we refer to [192]. But some of them do assumptions over the alphabet that could no be fulfilled by our grammar based application and that is because we could not compare them with our algorithm. The two assumption that excluded some of them were:

1. the size of the alphabet. Manzini and Ferragina's algorithm [156] and Yuta Mori's *libdivsufsort* [165] suppose a size of alphabet less than 256. In our approach, in each iteration we introduce a new non-terminal, so this bound is too tight.

2. it is possible that, after a replacement, a letter does not occur any more in the sequence because all its occurrences were inside the selected repeat. That is why we discarded algorithms that suppose a contiguous alphabet (like [154]).

The tests were executed on a 1GHz AMD Opteron processor with 4Gb of memory.

First, to have an idea of the complexity of the algorithm, we studied how the length of the sequence influences the execution time of the algorithm. From the large Calgary corpus, we extracted sequences of different lengths by considering successively bigger (by steps of 100 kilobytes) prefixes of the sequences. On each extracted sequence, we performed 250 iterations of selecting a random

[4] available at `http://www.irisa.fr/symbiose/projects/suffix_array_update`

repeat, replacing it over the sequence by a new character and updating the associated suffix array. Time (user + system time) required for updating the suffix array was reported, averaged over 5 different runs corresponding to 5 different random seeds. The same experiments, replacing the update algorithm by the from scratch construction algorithms of the suffix array by Kärkkäinen and Sanders (*K & S*), Larsson and Sadakane (*L & S*) and Zhang, Nong and Chan (*ZNC*) have been performed. The plots, shown in Fig. 3.8, confirm that the execution time of our updating algorithm is not directly correlated to the length of the sequence, and is significantly smaller than the execution time required by construction from scratch algorithms, especially when the length of the sequence increases.

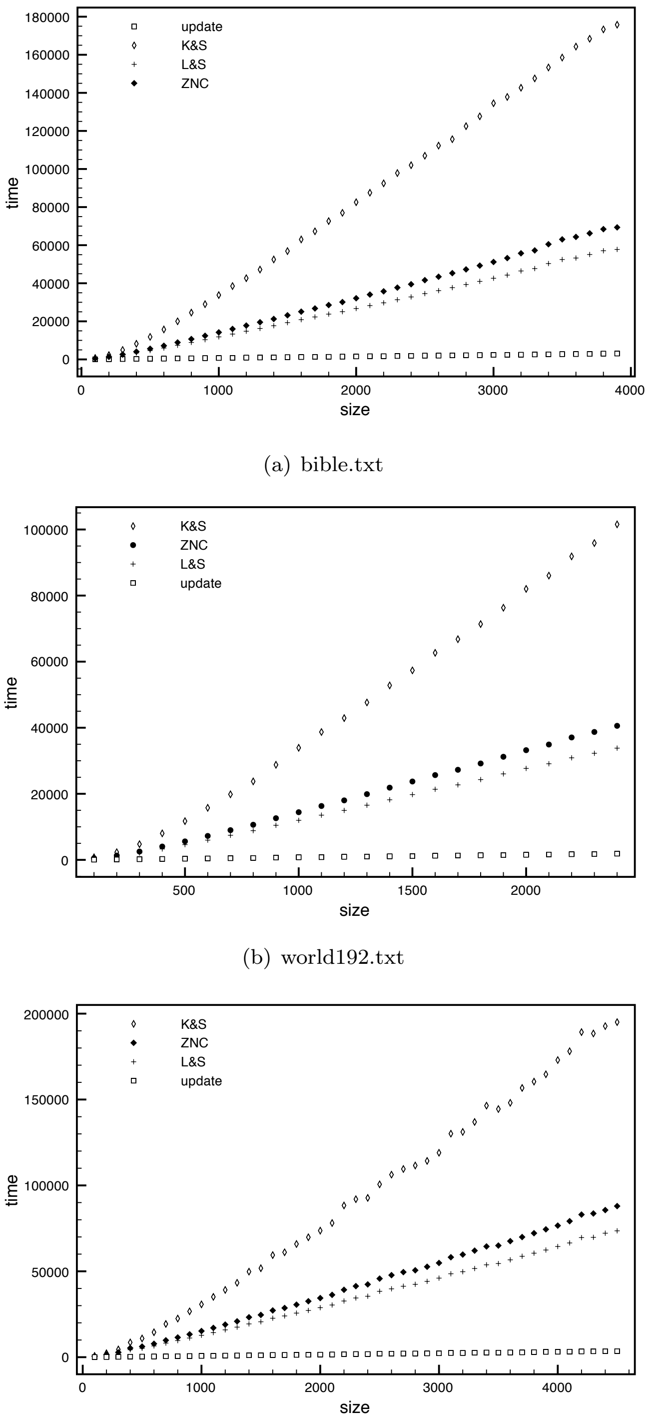
We present a more exhaustive evaluation and comparison on all the corpora using different strategies for the selection of the repeated word. In each test we performed 500 iterations of selecting a repeat, replacing it over the sequence and updating (or building from scratch) the associated enhanced suffix array. The different strategies for the selection of the repeat were:

- take a random one (using the same seed for the pseudo-random number generator),
- take the longest (ML strategy),
- take the one that covers the maximal number of positions (MC strategy).

Results are given in Fig. 3.3 (page 67). For each selection strategy, we measured time (user + system time) spent in updating ESA_{DL} with our algorithm (column *update*), and time spent in building *ESA* from scratch at each iteration with the three creation algorithms. For easier comparison, we only report the times given by the update algorithm and the ratios of the time spent by each of the three "from scratch" algorithms over the update algorithm. A ratio lower than 1 means that the from scratch algorithm was faster than the update. Time spent by a from scratch algorithm can be obtained by multiplying the time reported in the "update" column by the respective ratio.

Some of the files (notably `fields.c`, `grammar.lsp` and `xargs.1`) are too small to draw significant conclusions, but results are shown here for the sake of completeness. On the other files, results show that a significant speedup is usually achieved by using our algorithm. The main exception is the `ptt5` file from the Canterbury corpus (a fax image with very long zones of the same byte), probably because the rewriting of the selected repeats change a major part of the sequence. One can also remark that the ratio is less favorable when the repeat to replace is chosen according to the maximal compression strategy. On the one hand, in each iteration the resulting sequence is smaller and the suffix array creation from scratch for this sequence faster. On the other hand, there are more positions affected by the substitution and this affects the update algorithm.

These cases allow us to illustrate an intrinsic limit of the update approach when the length of the sequence is highly reduced by recoding: when the number of positions to update is larger than the number of positions in the resulting sequence, it may be worth adopting the from scratch construction algorithm (let us remark that the best algorithm to use can vary along the iterations). A solution to handle these extreme cases, would be to design a criterion on the



(a) bible.txt

(b) world192.txt

(c) E.coli

Figure 3.8: Large corpus: `bible.txt`, `world192.txt` and `E.coli`. Times are given in hundredth of seconds and the size in kilobytes (1 byte = 1 character).

repeat and its coverage to automatically choose the best algorithm to use (even at each iteration).

Space efficiency

The overall space complexity is $\mathcal{O}(n)$. In this section we analyse this bound more precisely.

Storing the *ESA* requires three arrays of integer, and the sequence is also stored in an integer array (recall that our approach is supposed to work with integer alphabets). On a 32-bit architecture, this equals $10n$ bytes (where n is the amount of symbols of the input). The ESA_{DL} structure needs to extend the *ESA* with two arrays of length n (*next* and *prev*). To implement the $\oplus$ and $\ominus$ operators, two extra arrays of size n were used. The recursion in Alg. 4 was implement with a queue. Like the queue of Alg. 7, it is bounded by n. An array of length n is used to check in constant time whether a couple $(i, depth)$ was already used and a last auxiliary array of size n is used as specified in section 3.4.3. To sum up, the memory needed by the algorithm is $40n$, plus at most $4n$ for the queues.

To see the practical memory usage, we measured it during the execution of the first type of test (Sect. 3.4.4) on `E. Coli`. In Fig. 3.9 we plotted the results for the update and the three from scratch algorithms. Note that in this case we measured the memory used by the whole process, while in Sect. 3.4.4 the time spent in searching the repeats was not taken into account.

differently from the results reported for the time efficiency analysis (were only the time spent for the update / build from scratch suffix array was measured), the memory usage here correspond to the memory used for the totality of the program (including the search of repeats).

It is worth noticing that in the from scratch approach, the memory usage has a peak in the first iteration and then decreases, while in the update approach the memory occupied by the ESA_{DL} remains the same. This cannot be observed in figure 3.9 because we only measured the maximal memory usage.

Each of the four curves shows a linear behavior. In general, both *L&S* and ZNC algorithm use $26n$ memory. This is consistent with the three arrays used for the *ESA* and the sequence, plus one to store the new sequence. The other $6n$ can be attributed to the algorithm that recovers the repeats. The fact that there is no apparent difference between both algorithms can be explained again by the fact that we measured only the maximal memory used. *K&S* uses much more memory, what is consistent with other reported results [165]. We can also observe that the the memory usage of our update algorithm reaches in this test the predicted $44n$ upper bound.

3.5 Summary

We presented in this chapter three ways of accelerating general IRR algorithms. The first consists in reducing the space of words to be considered for replacement to some subclass of all possible repeats. This is motivated also by the eventual possibility that these more restricted classes — especially for largest-maximal repeats — contain particularly meaningful constituents. Comparing the execution time with final grammar, we conclude that using maximal-repeat produces

	random				maximal length				maximal compression			
	update	speedup factor			update	speedup factor			update	speedup factor		
	time	K & S	L & S	ZNC	time	K & S	L & S	ZNC	time	K & S	L & S	ZNC
CANTERBURY												
alice29.txt	163	17.25	9.18	9.47	192	12.28	7.14	7.45	269	4.06	1.9	2.61
asyoulik.txt	131	16.11	8.47	8.78	127	13.6	8.34	8.69	182	4.76	2.23	2.88
cp.html	15	8.8	6.33	6.6	15	6.4	4.27	5	18	3.06	2.22	2.83
fields.c	6	6.33	5.17	6.17	8	2.38	2.63	3.88	3	6	2	5
grammar.lsp	3	1.67	1.67	3	0	div 0	div 0	div 0	0	div 0	div 0	div 0
kennedy.xls	1323	26.38	9.7	9.73	1230	29.24	11.22	10.87	1541	3.16	1.08	1.5
lcet10.txt	1248	5.92	3.7	6.07	522	31.51	12.35	14.01	749	7.76	3.02	3.97
plrabn12.txt	516	33.24	13.32	19.42	606	31.84	15.35	16.26	887	8.84	3.28	4.49
ptt5	588	38.53	15.06	4.8	696	7.65	5.32	3.41	1900	0.44	0.19	0.28
sum	42	5.57	3.6	3.9	34	5.5	2.91	4	28	2.93	1.71	2.18
xargs.1	6	4.17	1.5	4.83	2	3	1	4	2	2	1	7
LARGE												
bible.txt	5055	66.81	22.84	22.8	5168	64.39	22.5	21.96	10285	15.37	3.7	5.41
E.coli	5534	69.14	27.36	26.59	6307	53.46	24.03	21.8	14808	9.51	2.11	3.4
world192.txt	3084	65.06	21.75	22.12	3089	60.7	21.11	21.2	5573	16.28	4.54	5.8

Table 3.3: Update time and speedup factor with respect to each of the from scratch algorithm. Times are given in hundredth of seconds. A speedup factor lower than 1 means that the from scratch algorithm was faster than the update algorithm (all these cases are underlined).

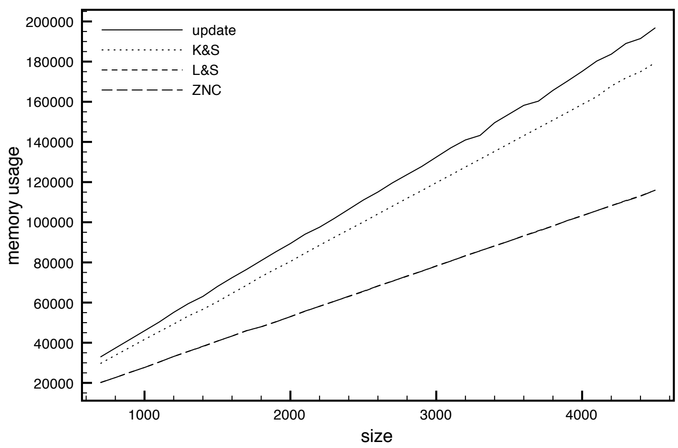


Figure 3.9: Memory consumption for `E. Coli`. Memory and size are given in kilobytes.

almost always grammars of the same size as the original IRR algorithm, while being faster.

A second improvement consists in approximating the real number of non-overlapping occurrences with the number of total occurrences. This improves considerably the execution time, and surprisingly also produces (slightly) smaller grammars. Combining this second improvement with the use of maximal repeats permits a generic $\mathcal{O}(n^2)$ implementation of the generic IRR framework, compared to the original $\mathcal{O}(n^3)$.

The third technique is different because it is an algorithmic improvement that does not change the output. We give a solution to a special kind of update of an enhanced suffix array. Our approach uses the specific internal order of suffix arrays to simultaneously update groups of adjacent indices and ensures that only indices to be modified are visited. This specific property of the suffix arrays allows to design an efficient update procedure which has been implemented and tested on classical corpora. The experimentation confirms that, in regard to the direct method reconstructing the suffix array, our approach enables significant speedup of the execution time of a factor up to 70 when choosing randomly a repeat to replace. The time improvement varies, and seems to depend mainly on the size of the left-context tree. This grows with both the average *lcp* value of the sequence, and the number of positions the chosen repeat covers. The results encourage to use such an approach in a greedy schema like IRR schema, where also be considered the occurrences on the right-hand side of previous introduced rules. Our discoveries in the next chapter however, made us realise the importance of being flexible with which occurrences to replace.

We focused here only on the final size of the grammar. Regarding the first two improvements, where different choices of words produces different final grammars, it could be interesting in future to analyse these changes not on the size of the final grammar, but on the structure obtained.

Chapter 4

Smaller Grammars

We concluded in Sect. 2.7 that the current state-of-the-art algorithm to find small grammars is IRR-MC. It belongs to the general IRR algorithm family and selects in each iteration a repeat that greedily reduces at most the size of the current grammar. Using different notions of size of a grammar, the same greedy approach has successfully been used in grammar-based algorithms like GREEDY [13, 14] and MDLCOMPRESS [84].

Our empirical results are completed by the theoretical ones in Charikar et al. [51]. The $\mathcal{O}(n/\log n)^{\frac{2}{3}}$ upper bound for IRR-MC given there may not look impressive, compared to algorithms specifically designed to achieve a good worst-case upper bound, but it should be kept in mind that this a loose bound, used to limit the behaviour of a much more general class of algorithms. Until now, no family of sequences could be exhibited on which IRR-MC achieves a non-constant approximation (the worst case being $5\log 3/(3\log 5) \approx 1.138$).

In this chapter we will present several algorithms that outperform IRR-MC. The main idea behind these algorithms is to separate the choice of which words will become constituents of the final grammar from the choice of which occurrences of these words will actually be replaced by non-terminals. The IRR permits flexibility in the first choice, but handles the second choice always in a greedy way (all occurrences in the normalised non-overlapping list are replaced), and preference is given to the first selected word. This is best exemplified in the following example:

Consider the sequence

$$xaxbxcx|xbxcxax|xcxaxbx|xaxcxbx|xbxaxcx|xcxbxax|xax|xbx|xcx,$$

This sequence exploits the fact that IRR algorithms replace all possible occurrences of the selected word. Let us define G^* as the following grammar:

$$\begin{array}{l} S \to AbC|BcA|CaB|AcB|BaC|CbA|A|B|C \\ A \to xax \quad\ B \to xbx \quad\ C \to xcx \end{array}$$

$|G^*| = 42$. Note that no IRR algorithm could generate G^* and, moreover, by brute-force search we can show that the smallest possible grammar that can be obtained with an IRR algorithm has size 46, resulting in an approximation ratio of 1.095. This is a global lower bound for *any* IRR algorithm[1].

We conclude therefore with the following theorem:

Theorem 4. *An algorithm that solves the Smallest Grammar Problem for all sequences does not belong to the IRR framework.*

Proof. We have seen that there exists a sequence s, such that $|IRR(s, f)|$ is greater than the size of a smallest grammar for s, for all possible choices of f. This proves our claim. □

Our approaches not only try to optimise the selection of constituents, but also the *parsing* of the sequence with these constituents.

This idea is not completely new. We have been able to find two references for this idea. At the end of a. shelat's master thesis [221, Sect. 5.3] he sketches the notion of *re-writing* (a similar notion to our minimal parsing) and *stable grammars*. Even before, Nevill-Manning et al. [178] noted that SEQUITUR "will not necessarily produce the smallest grammar possible. To do this would require finding two things: the best set of productions, and the best order in which they should be applied. The latter is called 'optimal parsing', and can be implemented by a dynamic programming algorithm." But to our knowledge, we are the first to formalise it and to study its importance and consequences.

First, we will formally define the problem of finding a minimal grammar given a fixed set of constituents (Sect. 4.1.1), similar to a recursive optimal parsing. In Sect. 4.2.1 we then define a search space for the Smallest Grammar Problem based on the MGP . We will introduce these two concepts along with the algorithms that use them. Part of this chapter is fruit of an INRIA/MINCyT collaboration and published/submitted previously [45, 46].

4.1 The Minimal Grammar Parsing Problem

4.1.1 Grammar Parsings and Minimal Grammar Parsings

Once an IRR algorithm has chosen a repeated word ω, it replaces all non-overlapping occurrences of that word in the current grammar by a new non-terminal N and then adds $N \to \omega$ to the set of production rules. In this section, we propose to perform a global optimisation of the replacement of occurrences, considering not only the last non-terminal but also all the previously introduced non-terminals. The idea is to allow occurrences of words to be kept (instead of being replaced by non-terminals) if replacing other occurrences of words overlapping them results in a smaller grammar.

Recall that the *constituents* of a grammar are the terminal strings that can be derived from the non-terminals of the grammar. We propose to separate the choice of which terminal strings will be constituents of the final grammar from the choice of how to parse the grammar with these constituents. First, let us assume that a finite set of constituents C is given and we want to find a minimal grammar that generates the language L and whose constituent set is C. In our case the language will be a singleton, but we will define first the general case.

[1]Note that this lower bound uses our definition of size, and can therefore not be compared with the lower bound given in Charikar et al. [51].

Assume that $C = \{\omega_1, \ldots, \omega_m\}$, and $L = \{s_1 \ldots s_\ell\} \subseteq C$. We need to be able to generate all constituents and for each constituent ω_i the grammar must thus have a non-terminal N_i such that $\omega_i = cons(N_i)$.

Therefore, we define a new problem, called **Minimal Grammar Parsing** (MGP) Problem. An instance of this problem is a tuple of sets of strings $\langle C, L\rangle$ with $L \subseteq C$. A **Grammar Parsing** of $\langle C, L\rangle$ is a context-free grammar $G = \langle \Sigma, \mathcal{N}, \mathcal{P}, S\rangle$ such that:

1. all symbols of all strings of L are in Σ: $\Sigma = \bigcup_{i=1}^{\ell} \Sigma(s_i)$
2. $L(S) = L$
3. for every other string $t \in (C \setminus L)$ there is one different non-terminal N that derives only t.

As we said, in the case we will consider from now on L will consist of only one string, s. The resulting grammar is therefore straight-line. A **minimal grammar** given $\langle C, L\rangle$ (or $\langle C, s\rangle$) is a grammar parsing G of smallest size $|G|$.

Note that the MGP problem is similar to the Smallest Grammar Problem, except that all constituents for the non-terminals of the grammar are given too. The MGP problem is related to the problem of static dictionary parsing [213] or optimal parsing (see Sect. 2.3) with the difference that the dictionary also has to be parsed. This recursive approach is partly what makes grammars interesting to both compression and structure discovery. For clarity, every time we use the term *minimal* grammar we will refer to a smallest grammar given a set of constituents, and we save the term *smallest* grammar to talk about globally smallest grammars.

As an example consider the sequence $s = ababbababbabaabbabaa$ and suppose the constituents are $\{s, abbaba, bab\}$ This defines the set of non-terminals $\{N_0, N_1, N_2\}$, such that $\text{cons}(N_0) = s$, $\text{cons}(N_1) = abbaba$ and $\text{cons}(N_2) = bab$. A minimal grammar parsing is $N_0 \to aN_2N_2N_1N_1a$, and $N_1 \to abN_2a, N_2 \to bab$.

The MGP can be solved in a classical way in polynomial time by searching for a shortest path in $|C|$ graphs as follows. Let the set of constituents be $\{s, \omega_1, \ldots, \omega_m\}$ and the language sequence s.

1. Let $\mathcal{N} = \{N_0, N_1, \ldots, N_m\}$ be the set of non-terminals. Each N_ℓ will be the non-terminal whose constituent is ω_ℓ.
2. Define m directed acyclic graphs $\Gamma_0, \Gamma_1 \ldots \Gamma_m$, where $\Gamma_\ell = \langle M_\ell, E_\ell\rangle$. If $|\omega_\ell| = k$ then the graph Γ_ℓ will have $k+1$ nodes: $M_\ell = \{1 \ldots |\omega_\ell| + 1\}$. The edges are of two types:
   (a) for every node i there is an edge to node $i+1$ labeled with $\omega_\ell[i]$.
   (b) there will be an edge from node i to $j+1$ labeled by N_m if there exists a non-terminal N_m different from N_ℓ such that $\omega_\ell[i:j] = \omega_m$.
3. For each Γ_ℓ, find a shortest path from 1 to $|\omega_\ell| + 1$.
4. The right-hand side for non-terminal N_ℓ is the concatenation of the labels of a shortest path of Γ_ℓ.

Intuitively, an edge from node i to node $j+1$ with label N_m represents a possible replacement of the occurrence $\omega_\ell[i:j]$ by N_m.

There may be more than one grammar parsing with minimal size. We suppose therefore any total order over grammars, and denote by $\mathbf{mgp}(C, s)$ the lowest minimal grammar parsing of $\langle C, s \rangle$.

In the case we consider, each constituent will be a substring of s. Note that in practice the graph Γ_0 contains therefore all the information for all other graphs because any Γ_ℓ is a subgraph of Γ_0. Therefore, we call Γ_0 the **Grammar Parsing graph** (*GP-graph*). See Fig. 4.1 for an example.

The list of occurrences of each constituent over the original sequence can be added to the graph at the moment it is chosen. The length of each constituent is bounded by $n = |s|$, so the complexity of finding a shortest path for one graph with a classical dynamic programming algorithm lies in $\mathcal{O}(n \times m)$. Because there are $m+1$ graphs, computing $mgp(C, s)$ is in $\mathcal{O}(n \times m^2)$.

4.1.2 IRR with Occurrence Optimisation

We can now define a variant of IRR, called *Iterative Repeat Replacement with Choice of Occurrence Optimisation* (IRRCOO) whose pseudo-code given in Algorithm 8. Different from IRR, what is maintained is a set of terminal strings, and the current grammar at each step is a Minimal Grammar Parsing over this set of strings. Recall that **constituent**$cons(\omega)$ gives the only terminal string that can be derived from ω (the "constituent").

The computation of the *argmax* depends only on the number of repeats, assuming that f is constant, so that its complexity lies in $\mathcal{O}(n^2)$. Like for IRR, the total number of times the while loop is executed is bounded by n. The complexity of this generic scheme is thus $\mathcal{O}(n \times (n^2 + n \times m^2))$, where $m+1$ is the number of constituents.

Algorithm 8 Iterative Repeat Choice with Occurrences Optimisation (IRRCOO)

IRRCOO(s, f)

Input: s is a sequence, and f is a score function on words

1: $C \hookleftarrow \{s\}$
2: $G \hookleftarrow G(\{S \to s\})$
3: **while** $(\exists \omega : \omega \hookleftarrow \underset{\alpha \in \hat{\mathcal{R}}(r(G))}{\arg\max}\ f(\alpha, G)) \wedge |mgp(C \cup \{cons(\alpha)\})| < |G|$ **do**
4: $\quad C \hookleftarrow C \cup \{cons(\omega)\}$
5: $\quad G \hookleftarrow mgp(C, s)$
6: **end while**
7: **return** $G(\mathcal{P})$

As an example, consider again the sequence used in the proof of Theorem 4. After three iterations of IRRCOO-MC the words xax, xbx and xcx are chosen, and a MGP of these words plus the original sequence results in G^*.

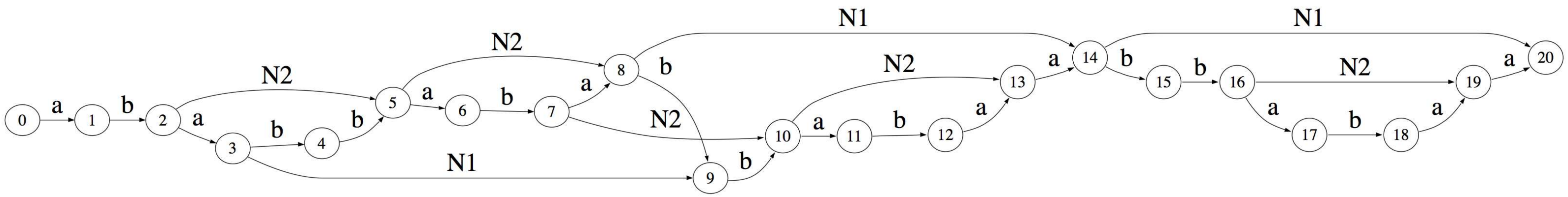


Figure 4.1: GP-Graph for $s = ababbababbabaabbabaa$ and additional constituents $\alpha_1 = abbaba, \alpha_2 = bab$.

4.1.3 Removing Costly Rules

The fact of re-arranging the non-terminals, optimising the size of the resulting parsing, may produce as side-effect that the use of some rules cost more than the gain they provide. Every rule that appears once or never is then *costly* and a rule that appears twice must have length at least 3.

Definition 12 (Costly Rule)**.** *Given a set of production rules $\mathcal{P}$, a rule $N \to \omega$ is a* **costly rule** *if* $(|\mathcal{L}_{r(G)}(N)| - 1) * (|\omega| - 1) < 2$.

So, given G, we denote by *clean*(G) the grammar where each costly rule $N \to \omega$ is eliminated and the occurrences of N replaced by ω.

Algorithm 9 presents the algorithm IRRCOOC (for IRR with Choice of Occurrence Optimisation and Cleanup): it is based on IRR, where a minimal grammar parsing and a cleanup is performed after each iteration. Recall that computing *mgp*(C, s) is in $\mathcal{O}(n * m^2)$ and every execution of line 7 reduces the size of the grammar by at least one. So, the worst-case complexity of IRRCOOC is again bounded by $\mathcal{O}(n^4)$ (m is bounded by n).

Algorithm 9 Iterative Repeat Replacement with Occurrence Optimisation and Cleanup (IRRCOOC)

IRRCOOC(s, f)

Input: s is a sequence, and f is a score function on words

1: $C \leftarrow\!\!\dashv \{s\}$
2: $G \leftarrow\!\!\dashv G(C)$
3: **while** $\exists\omega \ : \ \omega \leftarrow\!\!\dashv \underset{\alpha \in \hat{\mathcal{R}}(r(G))}{\arg\max} \ f(\alpha, G) \wedge |G_{\omega \mapsto N}| < |G|$ **do**
4:     $C \leftarrow\!\!\dashv C \cup \{\omega\}$
5:     **repeat**
6:         $C \leftarrow\!\!\dashv \{cons(N) : N$ non-terminal of G$\}$
7:         $G \leftarrow\!\!\dashv clean(mgp(C, s))$
8:     **until** G contains no costly rules
9: **end while**
10: **return** G

4.2 A Search Space for the Smallest Grammar Problem

The *mgp* procedure permits us to resolve the problem of finding a smallest grammar given a fixed set of constituents. With the IRCCOOC algorithm we introduced the idea of maintaining during its execution a set of constituents (C) from which a minimal grammar can be recovered. Here we go one step further with this idea and having resolved the problem of finding a minimal grammar given one set of constituents, we focus on finding a good set of constituents.

4.2.1 The Search Space

Consider the lattice $\langle 2^{\mathcal{R}(s)}, \subseteq \rangle$, where every node corresponds to a set of repeats of s. We then define a score function over the nodes of the lattice as $score(C) =$

$|mgp(\{s\} \cup C, s)|$. A **global minimum** C will be a node whose score is smallest: $score(C) \leq score(\theta)$ for all nodes θ.

For a given search space, we say that it is *correct* with respect to an optimisation problem P if each item of the search space that is a global minimum (with respect to a fixed score function) can be mapped to a solution of P. Conversely, the search space will be called *complete* if any solution of P can be mapped to a global minimum of the search space.

Theorem 5. *The lattice $\left\langle 2^{\mathcal{R}(s)}, \subseteq \right\rangle$ is a correct and complete search space for the Smallest Grammar Problem.*

Proof. Let $\mathcal{SG}(s)$ be the set of all smallest grammars for the sequence s. Similarly, we define $\mathcal{MGP}(C, s)$ the set of minimal grammars for $\langle C, s \rangle$. We will prove that:

$$\mathcal{SG}(s) = \bigcup_{C : C \text{ is global minimum of } \left\langle 2^{\mathcal{R}(s)}, \subseteq \right\rangle} \mathcal{MGP}(\{s\} \cup C, s)$$

To see the first inclusion ($\subseteq$), take a smallest grammar G^*. All strings in $cons(G^*)$ have to be repeats of s, so $cons(G^*) \setminus \{s\}$ corresponds to a node C in the lattice and G^* has to be in $\mathcal{MGP}(\{s\} \cup C)$. Conversely ($\supseteq$), all grammars of the right expression have the same size, which is minimal, so they are all smallest grammars. □

Because of the NP-hardness of the problem, it is fruitless (supposing $P \neq NP$) to search for an efficient algorithm to find a global minimum. We will present therefore an algorithm that looks for a local minimum on this search space. To define the algorithm, we first need some notation:

Definition 13. *Given a lattice $\langle \mathcal{L}, \preceq \rangle$, define:*

1. **ancestors**$(\eta) = \{\theta \neq \eta : \eta \preceq \theta \wedge (\nexists \kappa \neq \eta, \theta : \eta \preceq \kappa \preceq \theta)\}$
2. **descendants**$(\eta) = \{\theta \neq \eta : \theta \preceq \eta \wedge (\nexists \kappa \neq \eta, \theta : \theta \preceq \kappa \preceq \eta)\}$

The ancestors of node η are the nodes exactly "over" η, while the descendants of node η are the nodes exactly "under" η. A node η is a **local minimum** if $score(\eta) \leq score(\theta)$ for all nodes $\theta \in ancestors(\eta) \cup descendants(\eta)$.

4.2.2 The ZZ Algorithm

The ZZ algorithm, introduced by Carrascosa [43], looks for a local minimum, traversing the lattice in a hill-climbing way. In each step it selects the neighbour with the best score. But instead of inspecting all neighbours, it alternates two phases in which it inspects only the ancestors or descendants of the current node. This defines a path in form of zig-zag, therefore we name the algorithm *ZZ*. The two phases are *bottom-up* and *top-down*. The bottom-up can be started at any node, it moves upwards in the lattice and at each step it looks among its ascendants for the one with the lowest score. In order to determine which is the one with the lowest score, it inspects them all. It stops when no ascendants has a better score than the current one. As in bottom-up, top-down starts at any given node but it moves downwards looking for the node with the smallest score among its descendants. Going up or going down from the current node is equivalent to adding or removing a substring to or from the set of substrings in the current node respectively.

ZZ starts at the bottom node, that is, the node that corresponds to the grammar $S \to s$ and it finishes when no improvements are made in the score between two bottom-up–top-down iterations. Pseudo-code is given in Algorithm 10.

Algorithm 10 Zig-Zag algorithm

ZZ(s)

Input: s is a sequence

$\mathcal{L} \hookleftarrow \left\langle 2^{\mathcal{R}(s)}, \subseteq \right\rangle$
$C \hookleftarrow \emptyset$
while score(C) decreases **do**
  while $\exists C' \in \mathcal{L} : \left(C' \hookleftarrow \underset{C' \in ancestors(C)}{\arg\min} \text{score}(C') \wedge \text{score}(C') \leq \text{score}(C) \right)$ **do**
    $C \hookleftarrow C'$
  end while
  while $\exists C'$: $\left(C' \hookleftarrow \underset{C' \in descendants(C)}{\arg\min} \text{score}(C') \wedge \text{score}(C') \leq \text{score}(C) \right)$ **do**
    $C \hookleftarrow C'$
  end while
end while
return $mgp(C, s)$

Computational Complexity

In the previous section we showed that the computational complexity of computing the score function for each node is $\mathcal{O}(n \times m^2)$, where n is the length of the target string and m is the number of substrings in the node. At each iteration of top-down or bottom-up ZZ inspects up to $\mathcal{O}(n^2)$ neighbours, so each step upwards or downwards is made in $\mathcal{O}(n^3 \times m^2)$. No bottom-up (top-down) step can do more than n steps without increasing the score (a grammar with $n/2$ constituents has a size of at least n). Each of the bottom-up–top-down iterations decreases the size by at least one, so the final complexity o ZZ is in $\mathcal{O}(n^5 \times m^2)$.

4.2.3 Non-monotonicity of the search space

We finish this section with a remark on the search space. In order to ease the understanding of the proof, we will asume that the size of the grammar is defined as $|G| = \sum_{A \to \alpha \in \mathcal{P}} (|\alpha|)$. The proof extends easily if we consider our definition of size, but is more cumbersome (basically, instead of taking blocks of size two in the proof, take them of size three).

We have presented an algorithm that finds a local minimum on the search space. Locality is defined in terms of its direct neighbourhood, but we will see that the local minimality of a node does not necessarily extend further:

Proposition 6. *The lattice* $\langle 2^{\mathcal{R}(s)}, \subseteq \rangle$ *is not monotonic for function* $score(\eta) = |mgp(\{s\} \cup C, s)|$*. That is, suppose* η *is a local minimum. There may be a node* $\theta \supseteq \eta$ *such that* $score(\theta) < score(\eta)$*.*

Proof. Consider the following sequence:

$$abcd|cdef|efab|cdab|efcd|abef|bc|bc|de|de|fa|fa|da|da|fc|fc|be|be|ab|cd|ef.$$

The set of possible constituents is $\{ab, bc, cd, de, ef, fa, da, fc, be\}$, none of which has size longer than 2. Note that the digrams that appear in the middle of the first blocks (of size four) appear repeated twice, while the others only once. Also, the six four-size blocks are all compositions of constituents $\{ab, cd, ef\}$ (each of which is only repeated once at the end). Consider now the following grammar:

$$\begin{array}{lll} S & \rightarrow & aB^Cd|cD^Ef|eF^Ab|cD^Ab|eF^Cd|aB^Ef|B^C|B^C|D^E \\ & & |D^E|F^A|F^A|D^A|D^A|F^C|F^C|B^E|B^E|ab|cd|ef \\ B^C & \rightarrow & bc \\ D^E & \rightarrow & de \\ F^A & \rightarrow & fa \\ D^A & \rightarrow & da \\ F^C & \rightarrow & fc \\ B^E & \rightarrow & be \end{array}$$

of size 68, which is a smallest grammar given this set of constituents. Moreover, adding any of the three remaining constituents would increase the size of the grammar by one. But, adding all three of them would permit to parse the blocks of size 4 with only two symbols each, plus parsing the three trailing blocks with only one symbol. This means gaining 9 symbols and losing only 6 (because of the introduction of the new right-hand sides).

□

4.3 A Practical Algorithm

In Table 4.1 we summarise the size of the final grammars obtained with IRRCOO-MC, IRRCOOC-MC and ZZ on the Canterbury and DNA corpus, and compare them to the size of the grammars obtained with IRR-MC. In [44] we reported results using strategies other than MC with IRRCOO. As can be appreciated, each algorithm outperforms its previous version.

While ZZ proves to be very powerful, its big complexity ($\mathcal{O}(n^7)$)), makes it unfeasible even on the rather small corpora we use as benchmark. The efficiency and scalability concerns we analysed in Chapter 3 appear again. The IRRCOO and IRRCOOC frameworks are suitable to include the first two modifications we proposed to speed-up execution time. These are to consider only maximal repeats and to use the total number of occurrences instead of the size of a the normalised non-overlapping occurrence list. Because of the *mgp* algorithm, the last modification (in-place update of the underlying suffix array) seems more difficult to adapt and we did not include it here. Because the ZZ algorithms works different in the sense that it does not compute in each iteration the set of current repeats, we did not improve it with any of the proposed changes from Chapter 3.

We compared therefore the execution time of IRRCOOC-MC compared to the time spent by IRR-MC. Results over the DNA corpus can be appreciated in

Table 4.1: Result of the algorithms presented in this chapter on the DNA Corpus (a) and the Calgary Corpus (b). The size of the algorithms are given in percentage with respect to the size of the state of the art, IRR-MC (see Sect. 2.7). Cells marked with † refer to partial executions.

(a) DNA Corpus

sequence	IRR -MC	IRRCOO -MC	IRRCOOC -MC	ZZ	IRRMGP*
chmpxx	28,706	-2.86	-2.87	-9.35	-4.40
chntxx	37,885	-2.74	-2.75	-10.41	-4.85
hehcmv	53,696	-2.63	-2.69	-10.07†	-5.28
humdyst	11,066	-3.61	-3.62	-8.93	-3.86
humghcs	12,933	-0.50	-0.61	-6.97	-2.34
humhbb	18,705	-2.42	-2.43	-8.99	-4.07
humhdab	15,327	-2.11	-2.10	-8.70	-3.34
humprtb	14,890	-1.35	-1.36	-8.27	-3.45
mpomtcg	44,178	-1.90	-1.99	-9.66	-4.02
mtpacga	24,555	-2.37	-2.37	-9.64	-4.47
vaccg	43,701	-1.93	-1.95	-10.08†	-5.20
average	–	-2.22	-2.25	-9.19	-4.12

(b) Calgary Corpus

sequence	IRR -MC	IRRCOO -MC	IRRCOOC -MC	ZZ	IRRMGP*
alice29.txt	41,000	-3.05	-3.24	-8.05	-2.56
asyoulik.txt	37,474	-2.34	-2.38	-6.60	-1.80
cp.html	8,048	-1.33	-1.40	-3.49	-1.12
fields.c	3,416	-1.20	-1.41	-3.07	-1.11
grammar.lsp	1,473	-0.14	-0.14	-0.54	-0.14
kennedy.xls	166,924	-0.10	-0.11	-0.13	-0.09
lcet10.txt	90,099	-1.79	-1.85	–	-1.88
plrabn12.txt	124,198	-4.65	-4.72	–	-3.44
ptt5	45,135	-0.60	-0.84	–	-2.65
sum	12,207	-0.56	-0.57	-1.47	-0.82
xargs.1	2,006	-0.75	-0.75	-1.69	-0.45
average	–	-1.50	-1.58	-3.13	-1.46

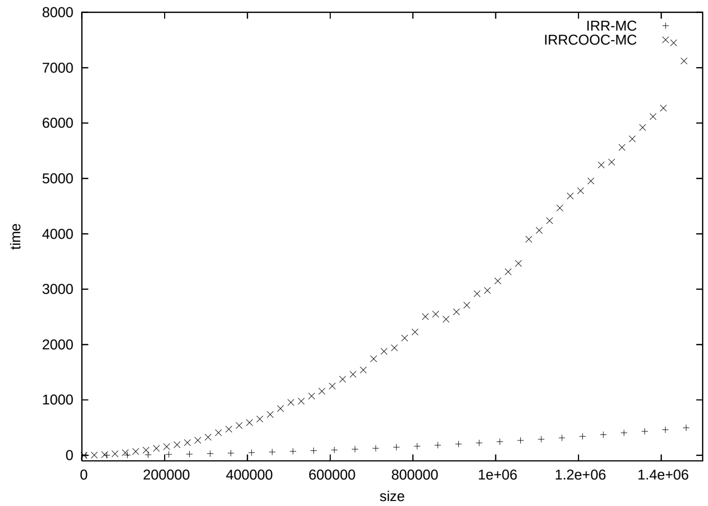


Figure 4.2: User time for consecutive prefixes of Eschericha Coli for the accelerated versions of IRR-MC and IRRCOOC-MC. Time is given in seconds and size in bytes

Table 4.2 IRRCOOC-MC finds grammar 1.96% smaller in average over the DNA corpus, needing five times more time, compared to the accelerated version of IRR-MC. (see Table 4.2). Unfortunately, it does not seem to scale up very well on bigger sequences. In Figure 4.2 we plot the user time required to execute IRR-MC and IRRCOOC-MC on successive prefixes of the Escherichia Coli genome. Both seems to grow as the square of the time (for the case of IRR-MC this can be better appreciated in Figure 4.3). The constant hidden in the complexity of IRRCOOC-MC however is much bigger than the one of IRR-MC, becoming unfeasible when applied to sequences bigger than the test corpus. Therefore, we present here a last algorithm, that is able to be executed on bigger sequences.

Analysing the time used by IRRCOOC in each instruction reveals that the bottleneck lies in the computation of $mgp(C, s)$. The way IRR choses its constituents is fast and quite direct, while optimising the occurrences of the constituents is much more expensive. Several choices of compromise are possible in order to reduce the number of times this optimisation step is performed. Here we propose to do it only at the end of an IRR execution and not at each iteration. Pseudo-code for this can be found in Algorithm 11. It consists of: run IRR, find a minimal parsing, throw away costly rules, and repeat this until no further improvement is made. By alternating IRR with a clean-up phase it reflects somehow the bottom-up–top-down phases of ZZ. We call this algorithm IRRMGP* because it can be seen as several applications of IRR-MC completed by a minimal parsing and cleanup.

Both the execution of IRR and the occurrence optimisation step reduces the size of the grammar by at least one. So, IRRMGP* is in $\mathcal{O}(n^4)$ too. However, we measured again the time needed on successive bigger prefixes of the Escherichia

Algorithm 11 IRR plus MGP (IRRMGP*)

IRRMGP*(s)

Input: s is a sequence

$G \mapsfrom G(\{S \to s\})$
while $|G| \neq IRR(r(G), f_{MC})$ **do**
  $G \mapsfrom IRR(r(G), f)$
  repeat
    $C \mapsfrom \{cons(N) : N$ non-terminal of G$\}$
    $G \mapsfrom clean(mgp(C, s))$
  until G contains no costly rules
end while
return G

sequence	IRRCOOC-MC		IRRMGP*	
	size	time	size	time
chmpxx	-2.53	5.62	-4.64	1.17
chntxx	-2.47	5.41	-4.74	1.14
hehcmv	-2.08	5.31	-5.16	1.09
humdyst	-2.61	3.58	-4.00	1.19
humghcs	-0.81	6.07	-2.34	1.15
humhbb	-1.66	4.59	-4.43	1.34
humhdab	-2.07	4.07	-3.41	1.12
humprtb	-1.16	4.39	-3.06	2.22
mpomtcg	-1.93	5.53	-3.85	1.13
mtpacga	-2.41	4.60	-4.36	1.20
vaccg	-1.78	6.36	-5.77	1.23
average	-1.96	5.05	-4.16	1.27

Table 4.2: Final grammar size and execution time of accelerated versions of IRRCOOC-MC and IRRGMP*. Grammar size are given as percentage with respect to the final grammar size obtained by the accelerated version of IRR-MC and time as ration with respect to the execution time of the same algorithm (see Table 3.2).

Coli genome. From the result in Figure 4.3 it can be appreciated that it has the same trend as IRR-MC and takes only slightly more time.

On the DNA corpus (Table 4.2) IRRMGP* obtains 4.35% smaller grammars on the classical test corpus, taking 27% more time compared to IRR-MC.

Thanks to its reasonable complexity, we were able to execute IRRMGP* on bigger sequences than those of the standard corpus. We chose model organisms from different kingdoms: *Phage lambda* (virus), *Escherichia coli* (bacteria), *Thalassiosira pseudonana* (chromista protist), *Dictyostelium discoideum* (amoebozoa protist), *Saccaromyces cerevisiae* (fungi), *Ostreococcus tauri* (alga), *Arabidopsis Thaliana* (plant) and *Caenorhabditis elegans* (nematoda). From the two protists (*T. pseudonana* and *D. discoideum*) we only took chromosome 1, for *A. Thaliana* we took chromosome 4 and chromosome 3 for *C. Elegans*. For all other cases the sequence corresponds to the whole genome. In each case, the analysed sequence was the flat DNA sequence, without annotations and where

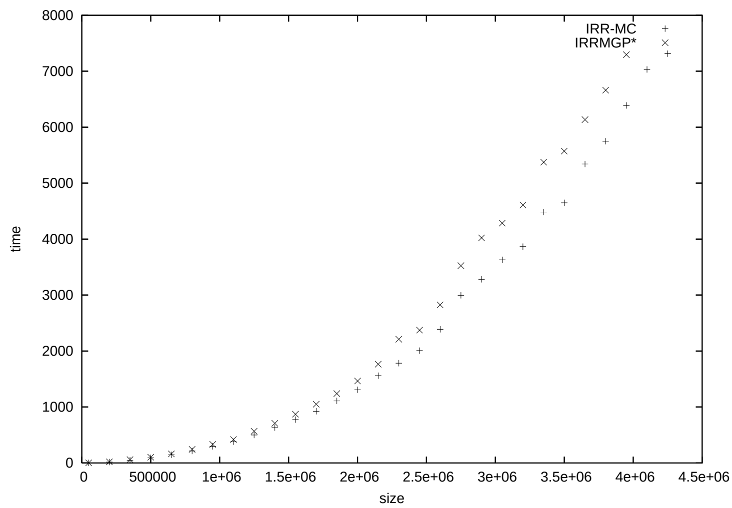


Figure 4.3: User time for consecutive prefixes of Eschericha Coli for the accelerated versions of IRR-MC and IRRMGP*. Time is given in seconds and size in bytes

any "N" was deleted. Table 4.3 shows the results. We report the size of the grammar returned by IRRMGP* and the improvement over IRR-MC which is close to 10%. In order to have a relative interpretation we also report the size of the IRRMGP* grammar divided by the length of the sequence. In general, we can see that this number becomes smaller (more redundancy is detected) when the sequence is bigger, but that it is not necessary correlated with the different kingdoms or classification of the analysed organisms. The average ratio on the classical DNA corpus is 0.23, in the same order as the ratio achieved on the rather small viral genome.

4.4 Summary

In this chapter we presented several algorithms that outperform state-of-the-art algorithms in the task of finding small grammars in practice. All these algorithms rely on the separation of two optimisations problem. The first is to find a minimal grammar given a set of constituents and defines what we named the Minimal Grammar Problem. We also presented an algorithm that solves this problem in polynomial time, the *mgp* algorithm. Each of algorithms presented in this chapter use this procedure. The other optimisation problem is to find an optimal set of constituents. The algorithms differ in the way they perform the search for this NP-Hard problem, ranging from the computational-intense ZZ to the much faster IRRMGP*.

In the next chapter we will analyse closer some of the possible applications of the Smallest Grammar Problem. In particular, analysing the uniqueness of

classification	sequence	length	IRRMGP*	$\lvert G\rvert/\lvert s\rvert$	gain
Virus	P. lambda	48,502	13,061	0.27	-4.25
Bacterium	E. coli	4,639,675	741,435	0.16	-8.82
Protist (Chromista)	T. pseudonana chrI	3,031,229	509,203	0.17	-8.15
Protist (Amoebozoa)	D. discoideum chrI	4,922,989	647,240	0.13	-8.49
Fungus	S. cerevisiae	12,156,679	1,742,489	0.14	-9.68
Alga	O. tauri	12,544,522	1,801,936	0.14	-8.78
Plant	A. Thaliana chrIV	18,582,009	2,561,906	0.14	-9.94
Nematoda	C. Elegans chrIII	13,783,317	1,897,290	0.14	-9.47

Table 4.3: Resulting grammar size for IRRMGP* on some model organisms. The last column shows the gain with respect to the size of the grammar of the accelerated version of IRR-MC.

the smallest straight-line context-free structure we will refer to the search space $\langle 2^{\mathcal{R}(s)}, \subseteq \rangle$ defined in this chapter.

Chapter 5

Applications

In the previous chapter we have seen how to obtain smaller straight-line grammars than the state of the art. Here, we revisit the three applications we described in Chapter 2, with the insights we obtained so far.

The first such application we will consider is Structure Discovery. We analyse a fundamental issue that concerns any use of the final parse tree of a smallest grammar (or an approximation of it) to discover a hidden structure. What happens if the smallest grammar is not unique? And if there are more than one of them, how similar are they? The splitting of the Smallest Grammar Problem we saw in the last chapter provides us with the tools to answer these questions. In Sect. 5.1 we analyse, both theoretically and on real sequences, the (non-)uniqueness of minimal grammars and their impact on the stability of the final structure. To measure the similarity we use standard metrics, and conclude this section proposing a new one that is more robust to minor changes.

Concerning Kolmogorov Complexity, we evaluate our IRRMGP* algorithm through clustering using the Normalised Compression Distance. Such an approach is parameterised by several options. We repeat two standard experiments from the literature, changing only the compressor function.

We worked mostly on applications for Data Compression. We mentioned already the importance of the data compression community in the field of smallest grammar (or grammar-based codes). Most of the reported results combines in some way the three steps we described in Fig. 2.2 (page 22). Nevill-Manning and Witten [172] for instance combine the grammar output by SEQUITUR with an implicit "marker" model and encode this afterwards with an 0-AAC. Results are comparable to *gzip* (it performs better on 6 of 14 files of the Calgary Corpus). We already saw the methods used by Yang and Kieffer [244, 245] (see Sect. 2.3.4) and their claims to compare similarly to the *PPM* family (though it is not clarified which version). In Sect 5.3, we analyse each of these steps separately. We pay special attention to the case of DNA compression, a field were grammar-based codes traditionally performed poorly, and introduce a compressor that outperforms any other grammar-based DNA compressors. In Sect. 5.4 we study the family of rigid patterns and how to include them in a straight-line grammar DNA compressor. This compressors achieves compression rate up to a 25% better than the previous best grammar-based compressor. The average compression ratio of this prototype on the standard test corpus is only 5% worse than the state of the art in DNA compression.

5.1 Structure Discovery

Since its origins [211], the problem of finding the smallest context-free grammar that generates a given sequence has been motivated by the desire of discovering the hidden structure of the object it describes. The example of results shown with SEQUITUR [171, 172, 173] where known structures over natural language, structured text and musical scores were unveiled are indicators that a small grammar may reveal interesting hierarchical patterns in real-life sequences.

In particular, a smallest grammar is the one that successfully extracts all possible redundancy that can be captured with the expression power of context-free straight-line grammars. Following Occam's Razor (see Sect. 1.3) this is a good candidate for the best explanation. However, it seems clear that some sequences may present more than one grammar whose size is minimal. In this section we first analyse the non-uniqueness of smallest grammars in theory, using for this the search space in form of lattice we defined in Sect. 4.2.1. We then study whether some stable structures are observed in practice among all minimal grammars (given a set of constituents). We present the results of experiments comparing the similarity of these grammars. Part of this section was realised in collaboration through an INRIA/MINCyT project and submitted for publication in 2010 [45].

5.1.1 Non-uniquenes of the Smallest Grammar: in Theory

Recall from Theorem 5 (page 75) that the lattice we defined is a complete search space in the sense that each smallest grammar is a minimal grammar parsing of a global minimal node. Here we will consider the number of such global minima. This is, the number of nodes whose minimal grammar parsing has a smallest size. The following lemma show that there may be an exponential number of these:

As in the proof of Proposition 6 and only to ease the understanding of the proof, we will use $|G| = \sum_{A\to\alpha\in\mathcal{P}}(|\alpha|)$ as the definition of grammar size.

Proposition 7. *Let* $n(k) = \max_{s:|s|=k}\left(\textit{number of global minima for } \left\langle 2^{\mathcal{R}(s)}, \subseteq\right\rangle\right)$. *Then,* $n(k) \in \Omega(2^k)$.

Proof. It is sufficient to find one family of sequences for which the number of global minima is exponential. Consider the sequence

$$s_k = a_1a_1|a_1a_1|a_2a_2|a_2a_2|\ldots|a_ka_k|a_ka_k = \prod_{i=1}^{k}(a_ia_i|)^2$$

over an alphabet of size $3k$. The a_i are single symbols. Recall that $|$ refers to a different symbol every time it appears. The set of repeated substrings longer than one is $\{a_ia_i, 1 \leq i \leq k\}$. Take any subset, and compute the (there is only one) smallest grammar with this constituent set. Adding any remaining constituents to this grammar reduces the length of the axiom rule by two, but does not reduce anything in the remaining rules, and adds two to the grammar size. The same happens with eliminating a constituent. So, any node of the lattice is a local minimum, and therefore a global one.

□

Next, we will suppose that the set of constituents is fixed and consider the number of minimal grammars that can be built with this set. This is, given a node η, try to bound $|\mathcal{MGP}(\eta)|$. As the following lemma shows, there are cases where the number of different minimal grammars can grow exponentially for a given set of constituents.

Proposition 8. *Let* $n(k) = \max_{s:|s|=k,\eta\subseteq\mathcal{R}(s)} (|\mathcal{MGP}(\eta \cup \{s\})|)$. *Then* $n(k) \in \Omega(2^k)$.

Proof. Let s_k be the following sequence $(aba)^k$ and let η be $\{ab, ba\}$. Each aba can be parsed in only one of the two following ways: aA or Ba, where A and B derives ab and ba respectively. Since each occurrence of aba can be replaced by aA or Ba independently, there are 2^k different ways of rewriting s_k and all of them have the same minimal size. □

5.1.2 Non-uniqueness of the Smallest Grammar: in Practice

In this section we analyse the number of minimal grammar on some of the sequence of our corpora, and we compare minimal grammar between them. As finding the smallest grammar is NP-Hard, we concentrate here on the node our best algorithm (ZZ, see Sect. 4.2.2) finds.

We would like to compute all the minimal grammar for a given node. Recall from Sect. 4.1.1 that, given a set of m constituents, the graphs Γ_i give a representation of all possible parses of constituent α_i. Here, we are interested in computing m subgraphs $\Delta_1 \ldots \Delta_m$ with the following two properties:

1. Let $\Gamma_i = \langle M_i, E_i \rangle$. Then $\Delta_i = \langle M_i, E'_i \rangle$ such that and E' is a subset of E.

2. Every path from node 0 to node $|M_i|$ over Δ_i is a shortest path for Γ_i.

Δ_i can be computed with a dynamic algorithm from Γ_i. At each node k, every income edge that is not part of a smallest path from 0 to k is eliminated. At the end, a filtering process is performed to eliminate every edge belonging only to paths that do not lead to node $|M_i|$ (see [45] for more details).

Using the Δ_i's, it is possible to compute all minimal grammars given a fixed set of constituents. In Table 5.1 we report the number of minimal grammars at the final node found by the ZZ algorithm. This number seems huge, and poses questions about how *similar* all these different grammars are.

sequence	`humdyst`	`asyoulik.txt`	`alice29.txt`
sequence length	38,770	125,179	152,089
grammar size	10,035	35,000	37,701
number of constituents	576	2,391	2,749
number of grammars	2×10^{497}	7×10^{968}	8×10^{936}

Table 5.1: Sequence length, grammar size, number of constituents, and number of grammars for different sequences.

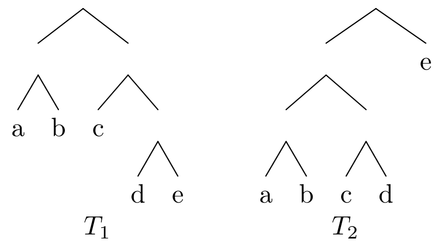


Figure 5.1: Two different trees with the same yield.

Similarity of Parse Trees

A straight-line grammar corresponds to one parse tree over the sequence, and comparison of parse trees is a recurrent task in fields like natural language processing. One of the problems in this area is, given a sequence of letters, words or part-of-speech tags, to automatically find the correct syntax tree. This is then compared to a manually annotated, "correct" tree, the *gold standard*. Despite of criticism [209], the Parseval metric [1], or an unlabelled variant of it [130, Section 2.2], seems still to be the standard metric.

Before we define this metric, we give some notation. Given a tree, a node is called **internal** if it has at least one daughter and **leaf** otherwise. The **yield of tree** T is the sequence composed by its leaves, read from left to right. Any node n defines a subtree T_n where n is the root. Clearly, $\mathit{yield}(T_n)$ is a substring of $\mathit{yield}(T)$. We define the **interval** of a node n to be the interval corresponding to the positions of $\mathit{yield}(T_n)$ in $\mathit{yield}(T)$. Finally the **bracketing** of a tree T is the set of intervals given by the internal nodes (except the root) of T. With the exception of the label of the nodes, the parse tree can be fully recovered from this set of intervals.

Consider for example Fig. 5.1. The bracket set of tree T_1 is $\{[0,4],[0,1],[2,4],[3,4]\}$ and the one of tree T_2 is $\{[0,4],[0,3],[0,1],[1,2]\}$.

The unlabelled version of the Parseval metric used to evaluate syntax tree is the Unlabelled F-Measure (the harmonic mean between the precision and recall). It is equal to the Dice coefficient of the bracketing of the evaluated trees:

Definition 14 (Dice coefficient)**.** *Given sets X, Y.*

$$\mathit{Dice}(X,Y) = 2 * \frac{|X \cap Y|}{|X| + |Y|}$$

In Fig 5.1, $\mathit{Dice}(T_1, T_2) = 0.5$.

In our experiments, there is no gold standard, so talking about precision and recall may be misleading. Instead of F_1-measure, we prefer therefore to refer to it as the Dice coefficient.

Dice-coefficient Similarity of Minimal Grammars

Seeing Table 5.1 it is clearly unfeasible to compute the exact similarity between all minimal grammars. We therefore sample some of the grammars and in order to have a uniform sample we proceed as follows: starting at the end node of each Δ_i, the sampling algorithm works its way back to the start node by repeatedly choosing one of the incoming edges with a probability weighted proportionally to the amount of shortest paths that go through that edge.

We computed the Dice-coefficient pairwise on a sample of 1,000 minimal grammars given the final node returned by ZZ. Results are reported, as percentages, in Table 5.2.

Sequence	`humdyst`	`asyoulik.txt`	`alice29.txt`
Dice mean	66.02	81.48	77.81
Dice standard deviation	1.19	1.32	1.52
Smallest value	62.26	77.94	73.44
Largest value	70.64	85.64	83.44

Table 5.2: Mean, standard deviation, smallest and largest values of similarity (Dice coefficient) given a uniform sample of 1,000 minimal grammars. All values are given in percentage.

Tracking the Difference

Until now we have seen that the number of different grammars given a local minimum over the lattice may be huge in real-life sequences. However, comparing them with Dice coefficient reveals that all these grammars present a rather stable structure. In this last two experiments we present here, we aim to discover where this difference lies.

In first place, we repeat the previous experiment, but filtering out those brackets that have a total size smaller than a given k. Note that the standard Dice coefficient gives the same weight to a bracket of size two than to longer brackets. For structure discovery, longer brackets seems to be more relevant. We denote this new measure as $Dice_k$. When $k = 1$, $Dice_k$ is equal to $Dice$, but for larger values of k more and more brackets are ignored in the calculation.

Table 5.3 reports the results for different values of k. For each sequence, the table contains two columns: one for $Dice_k$ and one for the percentage of the total brackets that were included in the calculation. As it can be seen, the Dice coefficient increases along k. This indicates that bigger brackets are found in most of the grammar, but it also shows that smaller brackets are much more numerous.

The second experiment considers the differences between the grammars on single positions. The objective of this experiment is to measure the amount of different ways in which a single position of the original sequence s can be parsed by a minimal grammar. For this we will consider the partial parse tree where only the first level is retained. Doing this for each minimal grammar, we compute for each position i the number of different subtrees it belongs to. This is equivalent to the number of edges in Δ_0 that starts at or before i and ends after i. If the number for one position is one for instance, this means that in all minimal grammars the same occurrence of the same non-terminal expands on this position.

On `alice29.txt`, 89% of the positions are parsed exactly the same way. A histogram for all values of different parses can be seen in Fig. 5.2. Note that the

k	`humdyst`		`asyoulik.txt`		`alice29.txt`	
	$Dice_k$	Brackets	$Dice_k$	Brackets	$Dice_k$	Brackets
1	67.32	100.00	81.50	100.00	77.97	100.00
2	71.11	45.99	88.26	50.86	83.70	53.14
3	75.93	37.54	92.49	29.57	87.94	32.42
4	82.17	15.69	95.21	19.85	89.60	22.01
5	88.51	3.96	96.35	11.78	88.88	14.36
6	95.46	1.24	97.18	8.23	89.45	9.66
7	98.38	0.44	97.84	5.72	91.50	6.44
8	99.87	0.19	97.82	3.83	92.78	4.30
9	100.00	0.06	98.12	2.76	92.37	2.95
10	100.00	0.04	98.35	1.88	91.87	2.10

Table 5.3: $Dice_k$ for different values of k. Values are given in percentage.

y-axis is in logarithmic scale. The number of positions reduce drastically if the number of parses is increased: only 10% of positions are parsed in two different ways, 1% in three and all others in less than 0.2%.

There were two regions that presented peaks on the number of different symbols. Both correspond to parts in the text with long runs of the same character (white spaces): at the beginning, and in the middle to indent a poetry.

While this experiment is only restricted to the first level of the parse tree, it seems to indicate that the huge number of different minimal parses is due to a small number of positions where different parses have the same size. Most of the positions however are always parsed the same way.

5.1.3 Structural Comparison of Sequences: a New Tree Distance Metric

The work in this section appeared in the Dagstuhl proceedings of the Seminar 10231 "Structure Discovery in Biology: Motifs, Networks & Phylogenies" [94].

We propose here a similarity metric between trees that ignores the labels of the nodes. It is inspired by the unlabelled version of the Parseval metric but our measure turns out to be much more robust to small changes of the brackets. The motivation itself is also different: while the goal of the Parseval metric is to evaluate how close a proposed tree comes to a gold standard, the objective of our measure is to compare different trees over possibly different sequences. This could then be used to perform a comparison of sequences based on their structure rather than on their sequential composition. The fact that the resulting function is a proper distance metric is useful and often necessary for applications that involve clustering for example.

It tries to extract all possible similarities, even if the matches of brackets are not perfect. In cases were (almost) nothing is known on the sequences, Dice coefficient may be too rigid and not be the right measure to rate similarity. It is useful to provide an objective score if the goal is to find the correct linguistic parse tree on small sentences and where small changes disturb completely the original meaning. But two parses $(\alpha)(abc)$ and $(\alpha a)(bc)$ for example would be judged as having a similarity of 0, regardless of the length of α. If α would be very long, it seems intuitive to assume that both parses highlight α as significant,

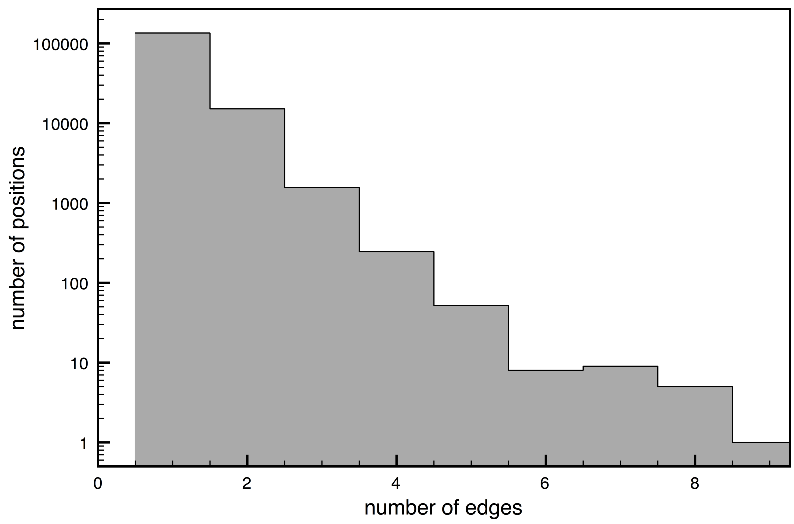


Figure 5.2: The X axis are the number of different symbols that expand to one position, the Y axis the number of positions that have this number of expansions. Note that Y is in logarithmic scale

but with an unsure right-border.

As a first step we focus on comparison between two brackets, which are represented as intervals. In order to measure how similar they are, we use the well-known Jaccard coefficient that gives a measure of similarity of two sets:

Definition 15 (Jaccard Coefficient)**.** *Given sets x,y,*

$$J(x, y) = \frac{|x \cap y|}{|x \cup y|}$$

Here, we suppose that the brackets are intervals over the integer line, so that each bracket defines a finite set of integers.

This gives a similarity coefficient between 0 and 1, 0 being completely different (empty intersection) and 1 total equality ($x = y$). The corresponding distance measure is $d(x, y) = 1 - J(x, y)$.

Proposition 9. *d is a proper metric, that is, it satisfies:* **non-negativity** *($d(x, y) \geq 0$),* **identity** *($d(x, x) = 0$),* **symmetry** *($d(x, y) = d(y, x)$) and* **triangle inequality** *($d(x, y) + d(y, z) \geq d(x, z)$)*

The first three are trivially true. For the triangle inequality see Lipkus [148].

In order to be able to compare the brackets of two trees (set of brackets) X and Y, we suppose an assignment function $f : X \to Y \cup \{\emptyset\}$. Moreover, we require this function to be injective, except possibly for $\emptyset$. This means, every bracket y from Y has at most one x from X such that $f(x) = y$. Let $R(f)$ denote the range of function f. Note that $R(f)$ may be only a subset of Y. The role of the empty set in the image is to permit to assign brackets from

X which otherwise would not be assigned. If $f(x) = \emptyset$ we refer to this as a **null-assignment**. Note that $d(x, \emptyset) = 1$, the maximal value for d.

We compare two set of brackets as follows:

Definition 16. $D_f(X, Y) = |Y \setminus R(f)| + \sum_{x \in X} d(x, f(x))$

This gives a penalty of a maximal distance for every bracket of Y to which no bracket of X was assigned. This is the symmetric part of assigning $\emptyset$ to a bracket of X.

In order to find the shortest distance between two trees, we are interested in finding the "best" possible assignment function. This is the function $f^* = \arg\min_f D_f(X, Y)$. Then, we define:

Definition 17 (Distance measure)**.** $D(X, Y) = D_{f^*}(X, Y)$

Symmetry

In order to prove symmetry, we define f^{-1} the inverse of a function in a non-standard way. Let f be as defined before. Then:

Definition 18 (Inverse function)**.** $f^{-1} : Y \to X \cup \{\emptyset\}$

$$f^{-1}(y) = \begin{cases} x & \text{if } y \in R(f) \text{ and } f(x) = y \\ \emptyset & \text{if } y \notin R(f) \end{cases}$$

So, each bracket from the image that was not assigned by f, receives a null-assignment of f^{-1}.

Proposition 10. $D_f(X, Y) = D_{f^{-1}}(Y, X)$

Proof.

$$\begin{aligned}
D_{f^{-1}}(Y, X) = & \sum_{y \in Y} d\left(y, f^{-1}(y)\right) + \left|X \setminus R\left(f^{-1}\right)\right| \\
= & \{\text{Definition 18}\} \\
& \sum_{x \in X \cap R(f^{-1})} d(f(x), x) + \sum_{y \in Y \setminus R(f)} d(y, \emptyset) + |X \setminus R(f^{-1})| \\
= & \{d(y, \emptyset) = 1\} \\
& \sum_{x \in X \cap R(f^{-1})} d(f(x), x) + |Y \setminus R(f)| + \sum_{x \in X \setminus R(f^{-1})} d(x, \emptyset) \\
= & \{\text{symmetry of } d\} \\
& \sum_{x \in X} d(x, f(x)) + |Y \setminus R(f)|
\end{aligned}$$

□

Now, if f does not assign brackets y from Y, the y is null-assigned by f^{-1}. So, if f^* minimises $D_f(X, Y)$, then $(f^*)^{-1}$ minimises $D_g(Y, X)$. We have as corollary:

Corollary 1 (Symmetry)**.** $D(X, Y) = D(Y, X)$

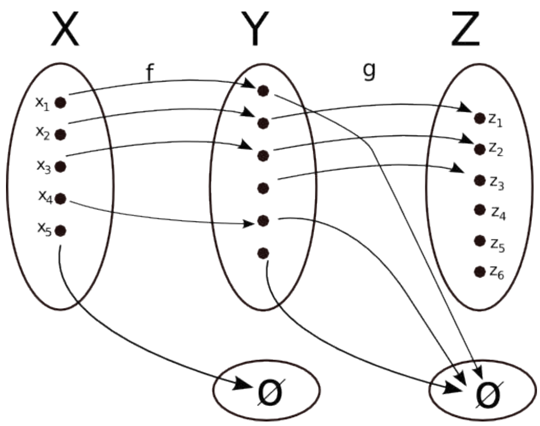


Figure 5.3: In this case $(g \circ f)(x_2) = z_1, (g \circ f)(x_3) = z_2$ and the rest are null-assignments.

Triangle Inequality

For the proof of the triangle inequality, we will proceed similarly as before, and redefine the composition of function.

Definition 19 (Composition)**.** *If $f : X \to Y \cup \{\emptyset\}$ and $g : Y \to Z \cup \{\emptyset\}$, then $g \circ f$ denotes the function:*

$$\begin{array}{lcl} g \circ f & : & X \to Z \cup \{\emptyset\} \\ (g \circ f)(x) & = & \begin{cases} g(f(x)) & \text{if } f(x) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

See Figure 5.3 for illustration. The intuition behind is that, if while going from X to Z over Y an x gets a null-assignment, then the final assignment is also null.

We will make use of the two following lemmas:

Lemma 6. $|Z \setminus R(g \circ f)| \leq |Z \setminus R(g)| + |Y \setminus R(f)|$

Proof. By Definition 19, $|Z \setminus R(g \circ f)| \leq |Z \setminus R(g)|$

□

Lemma 7. $\sum_{x \in X} d(x, (g \circ f)(x)) \leq \sum_{y \in Y} d(y, g(y))$

Proof.

$$
\begin{array}{rl}
& \sum_{x\in X} d\,(x,(g\circ f)(x)) \\
= & \sum_{x\in X, f(x)\neq\emptyset} d(x,(g\circ f)(x)) + \sum_{x\in X, f(x)=\emptyset} d(x,\emptyset) \\
\leq & \{\text{triangle inequality of } d\} \\
= & \sum_{x\in X, f(x)\neq\emptyset} \left(d\,(x,f(x)) + d\,(f(x),(g\circ f)(x))\right) + \sum_{x\in X, f(x)=\emptyset} d(x,f(x)) \\
= & \{\text{Definition 19}\} \\
= & \sum_{x\in X, f(x)\neq\emptyset} d(x,f(x)) + \sum_{x\in X, f(x)=\emptyset} d(x,f(x)) + \sum_{x\in X, f(x)\neq\emptyset} d(f(x),g(f(x))) \\
& \sum_{x\in X} d(x,f(x)) + \sum_{y\in R(f)\setminus\{\emptyset\}} d(y,g(y)) \\
\leq & \sum_{x\in X} d(x,f(x)) + \sum_{y\in R(f)\setminus\{\emptyset\}} d(y,g(y)) + \sum_{y\in Y\setminus R(f)} d(y,g(y)) \\
= & \sum_{x\in X} d(x,f(x)) + \sum_{y\in Y} d(y,f(y))
\end{array}
$$

□

Theorem 6 (Triangle Inequality)**.** $D(X,Y) + D(Y,Z) \geq D(X,Z)$

Proof. Suppose f, g, h are, respectively, the functions that minimise $D_e(X,Y)$, $D_e(Y,Z)$ and $D_e(X,Z)$.

$$
\begin{array}{rcl}
D(X,Z) & = & D_h(X,Z) \\
& \leq & \{\text{h minimises } D_e(X,Z) \text{, so in particular } D_h(X,Z) \leq D_{g\circ f}(X,Z)\} \\
& & \sum_{x\in X} d(x,(g\circ f)(x)) + |Z\setminus R(g\circ f)| \\
& \leq & \{\text{Lemma 7 and Lemma 6}\} \\
& & \sum_{x\in X} d(x,f(x)) + \sum_{y\in Y} d(y,f(y)) + |Z\setminus R(g)| + |Y\setminus R(f)| \\
& = & D(X,Y) + D(Y,Z)
\end{array}
$$

□

Computation

In this section we consider an algorithm to compute the measure D and analyse its computational complexity.

The Jaccard coefficient can be computed in constant time because we restrict the sets to be intervals. Given the assignment function f, $D_f(X,Y)$ is computable in $\mathcal{O}(n)$, where $n = \max(|X|,|Y|)$. Note that we only consider trees where every node has at least two daughters, so the size of the set of brackets ($|X|$) is linear in the number of leaves (the length of the yield).

The computation of f^* can be mapped to optimise an assignment problem and can be solved, for example, by the Hungarian algorithm [136], which is in $\mathcal{O}(|X|^3)$. So, $D(X,Y)$ can be computed in $\mathcal{O}(n^3)$, where $n = \max(|X|,|Y|)$.

Experimentations

In order to test our distance measure and compare it to existing metrics we want to see if it is capable to distinguish groups of similar trees. For this, we start with a small set of radically different trees. We then modify copies of these original trees and use a clustering algorithm based on the distance metric to regroup them. The nature of the change is always the same, but is parametrised by a random distribution.

Starting from k sets of brackets, we copy each of them m times with some modifications. We use $k = 3$: a left-branching tree ($\{(i, n) : 1 \leq i < n-1\}$), a right-branching tree ($\{(0, i) : 1 < i < n\}$) and a centred tree ($\{(i, n-i) : 1 < i < \lfloor n/2 \rfloor\}$). In our set-up, $n = 30$ and we generate $m = 32$ modified copies of each one, resulting at the end in 99 bracket sets (we keep the 3 original trees). The modifications are obtained by changing each bracket with probability p. A change consists in a shift of the bracket to the left or to the right (choosing randomly). Shifting a bracket $[a, b]$ by ℓ consist in replacing it by $[a + \ell, b + \ell]$. The value ℓ is given by a geometric distribution with parameter q (plus one, to ensure that the bracket is changed). After each change, all overlapping brackets are shorten to avoid overlap. In order to not give preference to any bracket, the order in which they are considered is determined randomly. Throughout our experiments we use $q = 0.5$ and different values for p. For the clustering algorithm, we compute the square distance matrix and use a k-medoid algorithm ($k = 3$), taking the cluster with lowest total sum of distances to the centre after 20 runs (each run starts with a random selection of centres).

Note that the Dice dissimilarity metric $(1 - D(X, Y)$ does not satisfy the triangle inequality and is inadequate to be used to compute a distance matrix. It is however closely related to the Tanimoto distance ($Dice(X, Y) = \frac{2 * J(X,Y)}{1 + J(X,Y)}$), so we compared our distance metric to the Tanimoto distance, but this time applied to set of brackets. In Fig. 5.4, we plot p (the probability that a bracket is changed) against the number of hits of the final cluster. Each point is the average over 250 runs, each run consisting in a copy-modify-cluster-count step. As it can be appreciated, as the probability of modification increases, the Tanimoto distance becomes less accurate to discriminate the correct clusters. This reveals the binary nature of a match in the Tanimoto distance: or a bracket matches or not. Our distance is more flexible: this reveals to be counterproductive if there are only few changes, but if p increases, it reveals to be very well suited to cluster the right groups. Both measures results in the same number of hits for $p = 0.5$, but from there on the number of hits using the Tanimoto distance decreases considerably. Our D distance continues to improve, getting more or less stable at 92 correct hits.

We presented a new distance metric to compare trees and prove that it is a proper metric. The aim of this metric is to compare sequences based on their tree structure. The advantage over previous approaches is its flexibility to compare trees that intuitively are highly similar, but where existing similarity metrics fail.

Our experiments show that this metric permits to distinguish groups of similar parse tree. Starting with a group of radical different trees, we modified each bracket slightly. When the probability of changing a bracket is greater then 0.5, our metric outperforms considerably a classical distance metric.

It would be interesting to analyse how this measures behave with respect to a *tree-edit distance*. Differently from the Parseval measure and ours, they are not based on the similarity between the yields of the nodes, but on the

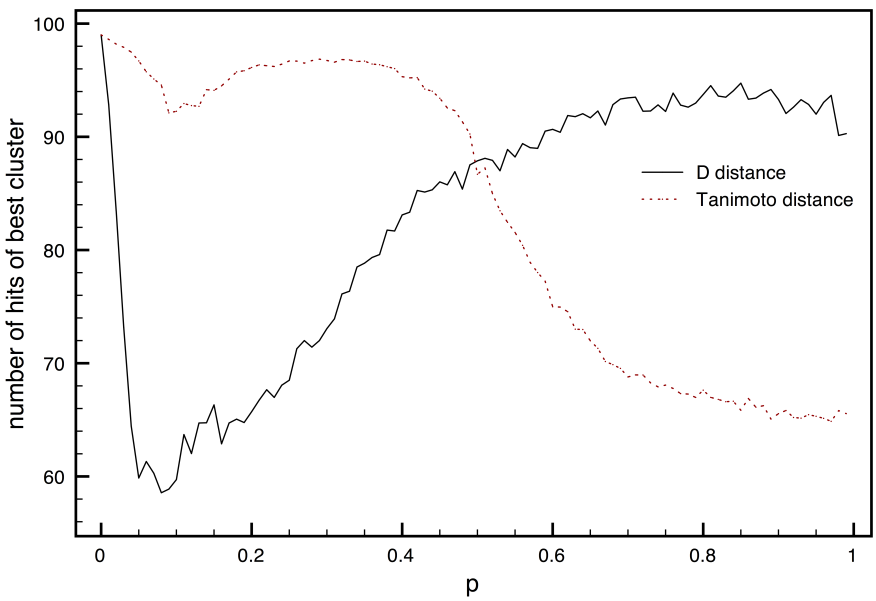


Figure 5.4: Number of true positives of the final cluster against probability of change of a bracket.

number of operations that are necessary to go from one tree to another one. The standard edit-distance [6, 29, 229] contemplates operations of insertion, deletion and renaming of nodes. In order to have a meaningful measure, an operation on edges that would permit to re-branch nodes (known as prune and regraft [7]) should also be considered.

5.1.4 Summary

Structure Discovery is possibly one of the most appealing and promising applications of the smallest grammar problem. Previous studies — particular with the Sequitur algorithm — has evidenced its potential with examples. However, a qualitative study that certifies a (semi-)automatic approach is missing.

We have analysed in this section a fundamental step toward such an approach, namely, the non-uniqueness of the smallest grammar. We proved that the number of smallest grammars can be exponential in the size of the original sequence, both because there may be an exponential number of constituents set that yields a smallest grammar or because there may be an exponential number of parses with one given constituent set. Our definition of the problem that decomposes it into a search for a set of constituents, and a parsing with this constituents, provided us with the tools (the Δ_i's graphs) to analyse the number of and similarity between minimal grammars. While the total number of different minimal grammars is huge they seem to be very similar between them. Moreover, most of the differences are due to the smaller constituents (which could be argued to be less interesting from a pattern-discovery point of view).

Also, there seems to be zones which concentrate most of the differences while other areas are much more stable. These stable constituents seems to be the most promising to unveil an eventual hidden structure.

Finally, we remark that an exact comparison as performed by standard tree similarity (or distance) measures may not be the right path for comparing parse trees of such big sequences. This applies particularly to DNA sequences were the boundaries of meaningful segments are not always stable. Therefore, we proposed a new proper distance metric over bracket sets that is able to detect similarity even if the trees are slightly modified. It would be interesting to define a normalised variant of this metric, and to use it in an efficient implementation that would reduce practical space and memory requirement of the Hungarian algorithm. This could then be used to perform similar experiments to those presented in the first part of this section.

5.2 Kolmogorov Complexity

The size of a smallest grammar provides an approximation to Kolmogorov Complexity. However, any such approximation is impossible to evaluate directly because of the non-computability of Kolmogorov Complexity (Theorem 2). Classical evaluations use an approximation of the Universal Metric (Def. 6, page 27), to perform classification or clustering tasks. Such an approach consists of several parameters: in first place a distance metric (like Normalised Compression Distance) and a compressor that approximates Kolmogorov Complexity. All pairwise distances are computed and the resulting distance matrix is used as input for a final algorithm that outputs, for example, a hierarchical cluster. Ferragina et al. [89] compare all possible combinations for three approximations of the Universal Metric, two phylogenetic tree construction algorithms (UPGMA and Neighbor Joining) and 25 compressors: gzip, bzip2, four variants of PPMd, Huffman, Arithmetic, Range coding, BWT+(MTF)+Run-length+[Huffman|Arithmetic|Range], BWT+Wavelet and Gencompress. For Arithmetic and Range encoding up to three variants where used, depending on how fast the model adapts to new statistics. These combinations were evaluated by the similarity of the resulting tree with a gold one, on six different data sets:

AA-15-DNA The *Apostolico* data set, consisting of 15 complete mitochondrial DNA genomes.

CK-36-PDB The *Chew-Keden* data set, consisting of 36 protein domains, in FASTA format (amino acid sequence).

CK-36-REL The same data set, but this time using their complete topological description (TOPS strings with contact map).

CK-36-SEQ The same data set, but this time in TOPS strings of secondary structure with contact map.

SP-86-PDB The *Sierk-Pearson* data set, consisting in 86 protein domains, in FASTA format (amino acid sequence).

SP-86-ATOM The same data set, but this time in ATOM lines from PDB entries.

The authors conclude with a recommendation of using NCD (or UCD, whose result is indistinguishable), UPGMA and PPMd, even if the performance of the different compressors varies largely with the type of data.

Cilibrasi and Vitanyi [63] use a tree construction algorithm designed by themselves, called the *quartet method* and implement it in their `CompLearn` tool[1]. In this same paper they present results of very wide range of experiments, with a particular emphasis on a phylogenetic tree reconstruction of mammals using their mitochondrial DNA. We repeat here this last experiment.

In both test, we used exactly the same conditions, but using our algorithms as compressors.

5.2.1 Biological Classification

We resume in Table 5.4 the results of using $C(x) = \mathit{IRRMGP}^*(s)$ in Def. 7 (page 27) and comparing against the gold tree given by the authors on their public website. In Table 5.4 we report exactly the same measures as Ferragina et al. [89]: F_1 measure (Dice's coefficient) for the CK-36 and SP-86 corpora, and the *partition distance*[2] for the AA-15 corpus. This should be compared with the corresponding Tables given by Ferragina et al. [89]. In general, our measure of compression works rather good, beating (or equaling) every other compressor for the CK-36-REL corpus. For the other two corpora of the CK-36 set, the result are in the upper third, loosing mostly against the PPMd family and performing mostly better than all the others. The overall result for the SP-86 set is rather poorly, with no combination scoring more than 0.6. The result of using IRRMGP* varies a lot: while it scores as bad as the worst on the ATOM corpus using the NJ algorithm, it is only outperformed by two compressors on the PDB+NJ combination. With respect to AA-15, our algorithm performs as good as the best compressor (PPMd-16).

In general, using the size of the grammar obtained by the IRRMGP* algorithm performs as good as the most advanced compression compressors analysed in Ferragina et al. [89]. A notable exception is on the SP-86 corpus, but this seems a general hard task where all of the compressors fail to achieve satisfactory results. In particular, as the SP-86-ATOM consists of tables with mainly numerical values, a general purpose-compressor may not be able to use all the available information.

5.2.2 Mammalian Phylogeny

Cilibrasi and Vitanyi [63] report a phylogeny tree using general compressors (zip, PPMZ, bzip2) and the CompLearn toolkit. They also compute all pairwise distance, but for the final binary-tree construction a new method, the *quartet method*, is used. This methods tries to optimise a score $S(T)$, where $S(T) = 1$ indicates that the tree T represents perfectly the distance matrix. We refer to their paper for details.

Using $|\mathit{IRRMGP}^*(s)|$ as approximation to Kolmogorov Complexity, and using the CompLearn toolkit (command `maketree`) we obtain the unrooted binary

[1] http://complearn.org/

[2] "takes in input the tree topologies of two alternative classifications of n species and returns a value ranging from 0 to $4n-10$. It is the number of clades in the two rooted trees that do not match and it is increasing with dissimilarity. When zero, it indicates isomorphic trees."[89]

CK-36-PDB	UPGMA	0.8676
	NJ	0.8824
CK-36-REL	UPGMA	0.9031
	NJ	0.8881
CK-36-SEQ	UPGMA	0.8849
	NJ	0.7418
SP-86-ATOM	UPGMA	0.5473
	NJ	0.5265
SP-86-PDB	UPGMA	0.5381
	NJ	0.5363
AA-15	UPGMA	4
	NJ	3

Table 5.4: Biological Classification on data-set used by Ferragina et al. [89] with *IRRMGP** to compute the distances. The reported measures are F-Measure (for CK-36 and SP-86) and partition distance for AA-15.

of Fig. 5.5. The tree is very similar as the one reported by Cilibrasi and Vitanyi, with the notable exception of the wrong position of *Cyprinus Carpio*, the common carp (which is not a mammal).

5.3 Compression with IRR

With respect to compression, recall our schematic representation of grammar-based codes in Fig. 2.2 (page 22) The inference process is completed with a transformation of the grammar into a linear sequence, and finally with an encoding into a bit stream. We have seen in Sect. 2.3 that, with respect to Step 2, Nevill-Manning et al.'s Marker method [56, 178] takes advantage of the number of non-terminals that appear only twice. Yang and Kieffer [244] use the knowledge that any rule will have right-hand side at least two and then concentrate on additional information to find a good model that performs the final Step 3 through an arithmetic coder. On another hand, *RNACompress* [149] (see Sect. 2.3.5) uses the parse tree of the secondary structure to encode the grammar.

The cases we mentioned are examples of three possible ways we studied of encoding a small grammar. Because the number of symbols in the grammar is likely to be very big, a dynamic alphabet could permit the same identificator to refer to different symbols, depending of the moment it appeared. A second strategy is to use the extra knowledge that the sequence that is encoded represents a parse tree. In this way, a more adequate probabilistic model could be defined that would produce a smaller bitstream. Finally, a step forward with this idea would send on one hand the topology of the parse tree, and then the single symbols. We will see these approaches in more detail in Sect. 5.3.1. The results presented there were obtained, partly, in collaboration with Matthieu Perrin from the ENS Cachan and we will refer to his final report [186, in french] for more details.

A completely different approach is to define an inference process that from the beginning is guided to obtain good compression (Step 1 of Fig. 2.2). The

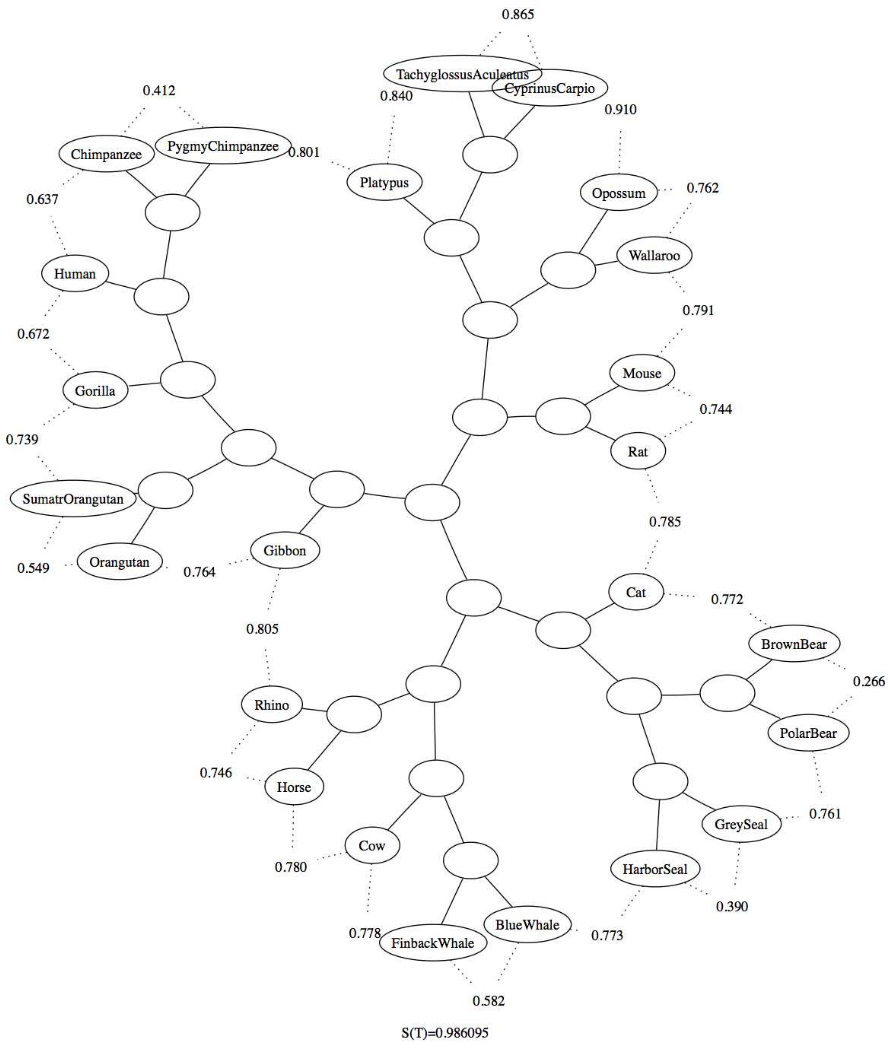


Figure 5.5: The unrooted binary tree after applying the `maketree` command of the CompLearn toolkit on a distance matrix obtained with *IRRMGP**.

grammar obtained this way may be very different from a grammar obtained searching for a smallest one [51, 158]. In Sect. 5.3.2 we do this and design an algorithm that generates a straight-line grammar which is optimised for compression. We include into this algorithm the possibility of considering also the complimentary strand of DNA and obtain a compressor that outperforms any other grammar-based DNA Compressor. We improve over this in Sect. 5.4 using inexact repeats and present a DNA compressor comparable to current state-of-the-art compressors.

5.3.1 Compressing Small Grammars

Given a small grammar, the alphabet size may be very big, but the actual number of symbols that are used at every right-hand side is much more reduced. So, our main concern in this section is to reduce the impact of this big alphabet.

Dynamic Alphabet

Because of the high number of symbols, we considered using a dynamic alphabet, so that the same identifier in the symbol stream could stand for different symbols from the grammar, depending when and where it appears. Our first approach was to maintain a dynamic set of *active alphabet* A. A is initialised empty and then $r(G)$ is read from left to right. At the first occurrence of a symbol s, it is assigned the identifier $|A|+1$, and added to A. Exactly after the last occurrence of a symbol s, a special symbol † is sent and s is removed from A. The right-hand sides of the remaining rules have to be sent in order of appearance of the first occurrence of the left-hand side. On our tests, the maximum size of A is about half the size of the original alphabet ($\Sigma \cup \mathcal{N} \cup \{|\}$)

M. Perrin proposes a similar approach [186, Sect. 4.2], but using Move To Front (MTF). The symbols are ordered on an array — called *order* — and $r(G)$ is again sent from left to right. Each time symbol s appears, the index i such that $order[i] = s$ is sent. The s is *moved to the front* of the array so that $order[0] = s$ and the remaining symbols between position 1 and $i-1$ are shifted one position to the right. The hope here is that the lower indices appear frequently, and that a adaptive arithmetic coder would concentrate probability on them.

None of the two approaches is particular to the grammar-based code framework, and the final size of the bit stream (coding them with a 0-order adaptive arithmetic coder) yield worse results then encoding directly $r(G)$. The second approach however, can be optimised because the order in which the right-hand sides can be sent can be altered in order to minimise the use of higher indices. The best approximation of this optimal solution [186, Section 6.2] are slightly better then the direct compression of $r(G)$, which is done with a $0-AAC$.

Specific Context Model for an Adaptive Arithmetic Coder

A possible reason for the failure of the most basic schema of the approaches presented above is that they were not designed to harmonise with Step 3, the final statistical encoding. Small grammars however, present several properties that may be exploited in order to define an ad-hoc probabilistic model. In irreducible grammars (see Def. 4, page 22) for example no substring is repeated.

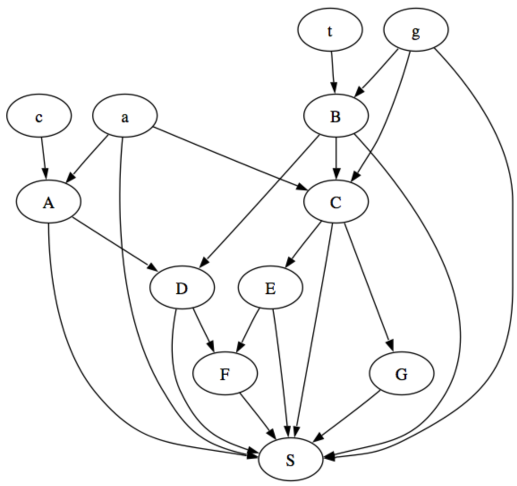


Figure 5.6: The directed acyclic graph corresponding to G_0.

M. Perrin uses a so-called *anti-dictionary* for a model that assigns null probability to symbol s if it has a left context of α and αs is a constituent of the grammar. Also, the power of an adaptive model can be exploited by presenting the right-hand sides by level of parse trees. First all rules consisting only of terminal symbols are sent, then those rules that also contains non-terminals of rules sent in the first group, and so on. On the DNA corpus, our tests report an improvement of up to 5% with respect to directly encoding $r(G)$ [186].

The Used-By Graph

Until now, the knowledge that $r(G)$ is not any sequence, but represents a parse tree over a sequence has not been used explicitly. It appears in the idea of ordering the rules by levels of this parse tree. However, the parse tree is a much richer structure than this. Suppose the following grammar G_0:

$$\begin{array}{lcl} S & \rightarrow & FAFaEBDgGCCG \\ A & \rightarrow & ac \\ B & \rightarrow & gt \\ C & \rightarrow & ag \\ D & \rightarrow & AB \\ E & \rightarrow & BC \\ F & \rightarrow & BDE \\ G & \rightarrow & CC \end{array}$$

We define the **directed acyclic graph** of a straight-line grammar G as $DAG(G) = \langle \mathcal{N}, E \rangle$, with $(N, N') \in E$ if N' appears in the right-hand side of the rule of N. $DAG(G_0)$ is shown in Fig. 5.6.

If the decoder would have knowledge of $DAG(G)$, then transmitting G could be done cheaply following a fixed order (breadth-first for example). Of course, it is not clear how to send the structure of the graph without spending more bits than are saved by limiting the alphabet in each node.

A similar approach is to consider the complete graph with nodes $\mathcal{N}$ and where each edge (N_i, N_j) is weighted with the cost of changing the alphabet of α_i to α_i (this could be, for example $|\Sigma(\alpha_i) \setminus \Sigma(\alpha_j)| + |\Sigma(\alpha_j) \setminus \Sigma(\alpha_j)|$. Finding a Hamiltonian path with lowest total weight on this graph would give the best order in which the rules should be sent.

Besides the interesting formalisation, all our preliminaries test showed that no gain (or a very small one) could be achieved, comparing to the straightforward encoding of $r(G)$.

5.3.2 An IRR Algorithm for Compression Purpose

Instead of trying to compress a given small grammar, we focus here on *generating* a grammar which is suitable to be compressed. This is similar to the objective of GREEDY [13], an IRR-like algorithm that defines a compression schema and selects in each iteration the repeat that reduces the most this schema.

Instead of this, we propose here to minimise the **empirical entropy** of the grammar.

Definition 20 (Empirical Entropy)**.** *Given a sequence s of size n, the empirical entropy of s is:*

$$\hat{H}(s) = \sum_{x \in \Sigma(s)} -|pos_s(x)| \log \frac{|pos_s(x)|}{n}$$

Then, in each iteration the algorithm IRR-S will select the repeat such that replacing all its normalised non-overlapping occurrence by a new symbol will result in a grammar G such that $\hat{H}(r(G))$ is minimal. As with IRR-MC the length and number of occurrences of the repeat plays a fundamental role, but the score function for IRR-S also takes into account the composition of the repeat.

With respect to the execution time, this means that in every iteration, for every repeat ω each symbol of the repeat has to be accessed (time $|\omega|$) and $\hat{H}$ be re-computed (time $|\Sigma \cup \mathcal{N}|$). This adds an extra $\mathcal{O}(n^2)$ to the execution of IRR, giving a $\mathcal{O}(n^4)$ factor. It seems to be possible to compute the content of each repeat in asymptotic time $\mathcal{O}(n)$, thus reducing the time complexity to $\mathcal{O}(|\Sigma \cup \mathcal{N}| \times n^2)$, but we did not optimise the algorithm for this. Our goal here was to find a straight-line grammar that represents a good compressor, and not optimising execution time.

We compare the result of the size of the final compressed stream (we compress $r(G)$ with 0-AAC) with the standard unix tools `zip` and `PPMd`. The alphabet $\Sigma(s)$ and order (we use the first $|\Sigma(s)|+1$ symbols for representing terminals and the sentinel) is supposed to be known by the decoder and therefore not sent. `zip` uses `compress` behind, which is an implementation of LZ78 (see Sect. 2.6.2). We use version 2.32 (June 19th 2006). The standard `zip` cannot be considered anymore as a state-of-the-art compressor, so we also compare to `PPMd`, an implementation (written by D. Shkarin and D. Subbotin) of a dynamic Markov encoding, a process that is similar to a variable context adaptive arithmetic coder. In Table 5.5 we report the bit per byte ratio of the three compressors on all sequences of the Cantebury corpus. IRR-S outperforms *zip* (except on sequence `sum`). We also report the number of rules found by IRR-S: a low number would indicate that most of the compression is done by the 0-AAC. It should be noted that the execution time of this non-optimised version of IRR-S is much

sequence	IRR-S bpb	IRR-S $\lvert\mathcal{N}\rvert$	zip bpb	PPMd bpb
alice29.txt	2.57	2,340	2.87	2.10
asyoulik.txt	2.86	1,892	3.14	2.34
cp.html	2.64	453	2.67	2.18
fields.c	2.36	293	2.41	1.98
grammar.lsp	2.77	109	3.15	2.35
kennedy.xls	1.40	1,444	1.61	0.96
lcet10.txt	2.25	5,083	2.72	1.90
plrabn12.txt	2.73	5,259	3.24	2.24
ptt5	0.82	1184	0.88	0.78
sum	2.86	683	2.75	2.51
xargs.1	3.34	135	3.73	2.89
average	2.42	–	2.65	2.02

Table 5.5: Result of IRR-S, *zip* and *PPMd* on the Canterbury corpus. bpb stands for bits (of the final compressed stream) per byte (of the original sequence).

longer (up to some hours) than *zip*. As expected, the compression ratio is well below *PPMd*.

Interestingly, the number of rules that IRR-S compute before stopping is much lower than the number of rules of IRR-MC. This is an argument for the observation that smaller grammars not necessarily compress better. Also, the kind of repeats identified by IRR-S differs from those identified by IRR-MC. While the repeats selected by this last one are mostly short strings which appears very frequent, IRR-S seems to prefer longer repeats. This joins the argument sketched in Bookstein and Klein [35] where different "measures of worth" for including a string in a dictionary are considered:

> "[...] a string may occur often simply because its components are expected to occur frequently. If the string occurs frequently only because its components do, no earnings accrue from reducing the string to a single object"

The authors then continue with an example that shows how, in a string generated by an i.i.d. source, to replace a substring with a new symbol will not achieve compression.

For the DNA corpus, we also take into account the reverse complement and call the resulting algorithm IRR^c-S. Instead of choosing different symbols for the non-terminal that represent a normal strand and the one that represent the complement strand, we use the same and disambiguate with a separate bit. Thus, each occurrence of a non-terminal adds one bit to the size of the grammar. The function that IRR^c-S then minimises is $\hat{H}(r(G)) + \sum_{N \in \mathcal{N}} |pos_{r(G)}(N)|$. At the end, we encode $r(G)$ with a 2-AAC (as recommend by Grumbach and Tahi [107]), hoping that the context could capture some more redundancy that escaped the grammar. Final relative compressed size and number of non-termianls are reported in Table 5.6. IRR^c-S produces in general less rules, and the final size is better than the grammar of IRR-S. This is particularly true for the two

sequence	IRR-S		IRR^c-S	
	bpb	$\lvert\mathcal{N}\rvert$	bpb	$\lvert\mathcal{N}\rvert$
chmpxx	1.8545	15	1.6780	17
chntxx	1.9539	17	1.6200	12
hehcmv	1.9747	31	1.8568	25
humdyst	1.9491	2	1.9326	3
humghcs	1.4721	641	1.4808	429
humhbb	1.9136	62	1.8723	46
humhdab	1.9272	101	1.8801	78
humprtb	1.9271	83	1.8890	68
mpomtcg	1.9631	129	1.9347	111
mtpacga	1.8690	45	1.8643	26
vaccg	1.8662	27	1.7744	13

Table 5.6: Compression results of IRR-S and IRR^c-S on the DNA corpus. $r(G)$ is compressed with 2-AAC

chloroplasts (`chmpxx` and `chntxx`) and the only exception to this is `humghcs` a highly-repetitive sequence. For `humdyst`, IRR-S finds only one additional rule (two in the case of IRR^c-S). Note however, that this is also the sequence with worst compression ratio for any of the present DNA compressors. Comparing with the state of the art and SLG DNA compressors (Table 2.1 and 2.3) gives some interesting insights. First, this rather simple algorithm outperforms any other grammar-based DNA compressor. Furthermore, to our knowledge this is the first time a compressor using a straight-line grammar outperforms a general-purpose compressor (like 2-AAC) on the whole DNA corpus. Second, the sequences where the difference between IRR^c-S and the best known DNA compressors is largest are exactly those sequences where the biggest number of rules are found (namely, `humghcs`, `humhbb`, `humhdab`, `humprtb` and `mpomtcg`). Therefore, a coding schema that would not be penalised so much by the extra number of symbols (the issue we addressed in Sect. 5.3.1) could cover this difference.

In the next section we will consider a different approach. By considering inexact repeats, we will try to reduce even more the final length of $r(G)$ without need of introducing more rules. We aim to mitigate the cost of introducing another symbol by permitting this symbol to cover more occurrences. Of course, the other side of the coin is that the decompressor has to have enough information to be able to decode in a lossless way each of these inexact occurrences.

5.4 Lossless DNA compression with Rigid Motifs

Besides the frequent occurrence of complementary repeats, another exploited property of DNA is the existence of inexact repeats. Non-exclusively statistical state-of-the-art DNA compressors like DNACompress [55] look for interesting inexact repeats and replace them before encoding the resulting sequence. The most used definition of *similar* makes use of the Hamming Distance or Edit Distance. Both distances require to specify the position where the edit (or deletion/insertion) occurs. This could easily became very large to be specified in bits. Therefore, we took a different focus and looked for *rigid patterns*. In this thesis, we will not consider any other kind of pattern, so we will use as interchangeable the terms *rigid patterns* and *motifs*.

Reflecting Sect. 3.2, we will first give a review of motifs, maximal motifs and irredundant motifs (Sect. 5.4.1). We give a new formalisation of irredundant motifs, which permits to implement easily an algorithm to compute them. In Sect. 5.4.2 we comment on the use of rigid motifs in a straight-line grammar and define a new algorithm that compresses well DNA sequences.

5.4.1 A Taxonomy of Rigid Motifs

The discovery of patterns from a given sequences is a major area of data mining, and has important applications in a wide range of domains like bioinformatics, musical analysis and natural language processing. In order to find those that are considered interesting it is sometimes necessary to consider *all* patterns. So, for such a search to be computationally feasible, the definition of what is a pattern must be a balance between its expression power and the total number of them that may exist in the sequence.

We already have seen (Sect. 3.2) how this leads to the definition of *maximal repeat* in the case of exact repeats. For several applications however, specially those related to genetic sequences, exact repeats are not enough to capture meaningful patterns. A possibility is to include a joker or "don't care" symbol in the pattern. This don't care symbol matches any other symbol, and permits to capture patterns that escape the universe of exact repeats. Unfortunately, the number of *motifs* can be exponential with respect to the size of the sequence. Even extending the notion of maximality does not improve this upper bound. But in 2000, Parida et al. [184] introduced the concept of *basis*, as a set of motifs such that all other motifs can be generated mechanically from them. Different basis have been proposed (see [187] for an overview), but in the last years a consensus seems to have been reached. The basis of *irredundant* or *tiling* motifs have the attractive property that their total number and the total number of their occurrences is bounded by n.

Very recently, several paper addressed possible applications of this type of motifs over genetic sequences. So, the use of irredundant motifs is used to classify proteins [72]. Other successful application combine the definition of maximal motif with statistical measures [20] or a ratio that bounds the number of don't cares [105].

Definitions

We present the definitions for motifs and recall a known lemma for reference.

We extend the alphabet Σ with an additional "don't care" symbol, denoted by $\bullet$ not contained in Σ and that matches any symbol. A symbol that is not $\bullet$ is called a **stable symbol**.

Definition 21 (motif)**.** *A motif is an element of* $\Sigma \cup \Sigma(\Sigma \cup \{\bullet\})\Sigma$. *Note that a motif cannot start or end with a don't care.*

If x is a motif, $inf(x)$ is x concatenated with infinite don't care symbols.

Definition 22 ($\preceq$)**.** *For $a, b \in \Sigma \cup \{\bullet\}, a \preceq b$ holds if $a = \bullet$ or $a = b$. This relation extends to strings: Let $x, y \in (\Sigma \cup \{\bullet\})^*$. $x \preceq y$ if $inf(x)[i] \preceq inf(y)[i]$ for any $i \geq 0$.*

Definition 23 (Occurrence of a motif)**.** *x **occurs** in y at position d, if $x \preceq y[d..]$. In this case, we say that y **implies** x. As for the case of exact repeats, we define, for the sequence s, $pos_s(x)$ (or just $pos(x)$ if s is clear from the context) as the set $\{i_1, \ldots, i_k\}$ such that m occurs in s at all i_j. If $i \in pos(m)$, then the tuple $\langle m, i\rangle$ is called an occurrence (of m).*

By $pos(x) + d$, we denote the set $pos(x)$ shifted by d positions: $\{i + d : i \in pos_x\}$.

Definition 24 (Maximal Motif)**.** *A motif x is a maximal motif, if for all motifs y that implies x, there is no d such that $pos(y) = pos(x) + d$.*

Finally, a maximal motif is said to be **redundant** if its occurrences can be obtained by the union of occurrences of other maximal motifs.

Definition 25 (Irredundant)**.** *Let x be a maximal motif. x is irredundant if, for every maximal motifs $y_1, \ldots, y_k$ and positive integers $d_1, \ldots, d_k$ such that $pos(x) = \bigcup_{i=1}^{k} pos(y_i) + d_i$ then $x = y_j$ for some j.*

An important concept in algorithms that retrieve irredundant motifs is the one of **autocorrelation**. We denote as s_k the k-th (for $1 \leq k < |s|$) autocorrelation of s, defined by:

$$\hat{s}_k[i] = \begin{cases} s[i] & \text{if } s[i] = s[k+i] \\ \bullet & \text{otherwise} \end{cases}$$

for all $i \in [0, |s| - k - 1]$. s_k is $\hat{s}_k$ after the removal of all leading and trailing don't cares. Note that s_k may be empty. Every non-empty autocorrelation defines two occurrences of this motif: ℓ and $\ell + k$, where ℓ is the position of the first solid symbol of $\hat{s}_k$. We denote by $\mathcal{M}$ the set $\{\langle s_k, \ell\rangle, \langle s_k, \ell + k\rangle :$ s_k is not empty and ℓ is the position of the first solid symbol of $\hat{s}_k\}$.

Pisanti, et al. [188] prove the following about autocorrelations:

Proposition 11 (Autocorrelations)**.**

1. *if s_k is not empty, it is a maximal motif*

2. *every irredundant motifs of s is an autocorrelation of s*

3. *$\langle x, i\rangle \in \mathcal{M}$ for all irredundant motif x and $i \in pos(x)$*

As they are $n - 1$ autocorrelations, the linear bound of the irredundant motifs and of the total number of their occurrences is a direct consequence of Proposition 11.

Alternative Characterisation

The definitions of maximal and irredundant motifs are given with respect to their set of occurrences. Here we propose an alternative characterisation, which will provide the basis for our algorithm afterwards. Our characterisation is based on the motif itself, instead of the position where it occurs. This avoids to compute the positions of the tiling motifs (the y_i's in Def. 25). The new characterisation permits also to make an intuitive parallel of maximal and irredundant motifs with their exact repeat (motifs without a don't care) counterparts.

As pointed out by Apostolico and Parida [15] maximal motif is intuitively a motif that cannot be made more specific without loosing one of its occurrences. By "made more specific", we mean expanding it to the right or to the left, or by changing a don't care into a solid symbol. Note that this is equivalent to Def. 23.

Theorem 7 (Characterisation of a Maximal Motif)**.** *A motif x is maximal iff for all motifs y that implies x, $|pos(y)| < |pos(x)|$.*

Proof. If y implies x, then each time y occurs, x does too; so $|pos(y)| \leq |pos(x)|$. The only-if part is then trivially true. For the if part, note that if x occurs in y, then it does so always with the same offset (the d from Def. 23), no matter the occurrence of y. So if y implies x, then $pos(y) \subseteq pos(x) + d$. This proofs the lemma. □

A similar approach has been taken by Ukkonen [234] to define maximal motifs: there two motifs are in the same equivalence class if they have the same set of occurrences (with an eventual offset). Maximal motifs are then those who have the maximal number of stable symbols in their equivalence class.

The characterisation of Theorem 7 is the intuitive counterpart of exact maximal repeats in the case of motifs. A maximal repeat x is an exact repeat such that any other exact repeat y that contains x appears less times. Note that in the case of exact repeats, a repeat can be made more specific only in expanding its length.

In order to give a characterisation of irredundant motifs in terms of implications, an intermediate step is necessary:

Definition 26 (Coverage)**.** *Let m and m' be motifs. m'* **covers** *m if $pos(m') + d \subseteq pos(m)$ for some $d \geq 0$. Occurrences $\langle m, i \rangle$ for $i \in pos(m') + d$ are said to be covered by m'.*

This leaves directly to

Lemma 8 (Characterisation 1 [185])**.** *A motif m is irredundant iff there is at least one occurrence $\langle m, i \rangle$ not covered by any other motif.*

Lemma 9 ([22, 185])**.** *Let m and m' be maximal motifs. m' covers m iff m' implies m.*

Note that the if part holds only for maximal motifs. If m' implies m, we say that the occurrence $\langle m, i \rangle$ is implied by $\langle m', i' \rangle$ if $\langle m, i \rangle$ is covered by m' and $i' \in pos(m')$.

Now it is easy to show that:

Theorem 8 (Characterisation 2)**.** *Maximal motif m is irredundant iff there is at least one occurrence $\langle m, i\rangle$ that is not implied by any other maximal motif.*

Proof. Directly from Lemmas 8 and 9. □

We call $\langle m, i\rangle$ a redundant occurrence if there is an occurrence $\langle m', i'\rangle$ that implies it. Note that all occurrences of redundant maximal motifs are redundant, and even some of the occurrences of an irredundant motif may be redundant.

It is worth to underline the similarity of this characterisation with the definition of largest-maximal repeats (see Def. 11).While the number of maximal repeats is linear, those of maximal motifs can be exponential. In the opposite, the number of occurrences of irredundant motifs are known to be linear while the number of occurrences of largest maximal repeats in the worst case is lower-bounded by $\Omega(n^{\frac{3}{2}})$ (see Proposition 4).

A Simple Algorithm to Compute Irredundant Motifs

Before introducing our algorithm, we give a short review of existing algorithms to compute the irredundant motifs:

Pisanti et al. [188] propose an algorithm, based on a filtering step of the autocorrelations (see Theorem 11). This filtering step consist in discard those motifs for which the y_i's from Def. 25 could be found. But for this, it is necessary to compute the occurrence set for all autocorrelations. To achieve this, the authors use the Fisher-Peterson algorithm based on a Fast Fourier Transform to compute boolean products. This defines the complexity of the algorithm (n applications of this algorithm), $\mathcal{O}(\log|\Sigma| n^2 \log n)$.

Pelfrêne et al. [185] propose another approach which also works for quorums different then two (the quorum of a motif is the size of its occurrence set). For the case of a quorum of 2 (the case we are considering), they also compute all autocorrelations and all their occurrences with the Fisher-Peterson algorithm. The filtering step is done using an alternative characterisation (Lemma 8). Again, it is the Fisher-Paterson algorithm who defines the complexity: $\mathcal{O}(|\Sigma| n^2 \log n)$

Apostolico and Tagliacollo [17] improve this bound to $\mathcal{O}(|\Sigma| n^2)$. The general schema is very similar to the two previous, but they are able to find all occurrences of all autocorrelation in time $\mathcal{O}(n^2)$ if the alphabet is binary.

Here we propose an algorithm that is not based on the occurrence list of the autocorrelations. Instead, it is based on the motifs itself and finds occurrences that fulfils Theorem 8. The main advantage of this algorithm does not lie in his complexity, which can be bounded by $\mathcal{O}(n^3)$, but its simplicity.

Our Algorithm As in the previous approaches, we first compute the autocorrelations of s. This can be done in $\mathcal{O}(n^2)$, and results in the set $\mathcal{M} = \{\langle x_1, o_1\rangle, \ldots, \langle x_m, o_m\rangle\}$ of occurrences, where m is bounded by n (see Propostion 11). All occurrences of irredundant motifs are in $\mathcal{M}$ (Propostion 11), so we must filter the occurrences of those motifs that are redundant. By Theorem 8 this means to filter occurrences $\langle x, i\rangle$ such that there exists $\langle y, j\rangle$ that implies $\langle x, i\rangle$. But because the relation *implies* is transitive, for each redundant occurrence $\langle x, i\rangle$, there must be at least one irredundant occurrence that implies it. So, if $\langle x, i\rangle$ is redundant, then there exists $\langle y, j\rangle$ in $\mathcal{M}$ that implies it.

Recall that if $\langle x, i\rangle$ implies $\langle y, j\rangle$, then there exists d such that $y[k] \preceq x[d+k]$ for all k. This means that for all positions of s where x is stable, y must be stable too. Thereafter, we define the set of stable positions of an occurrence: $stable(\langle x, o\rangle) = \{i_1, \dots, i_m : \text{ s.t. exists } \ell, i_j = o + \ell \text{ and } x[\ell] \neq \bullet\}$. Then $\langle x, i\rangle$ implies $\langle y, j\rangle$ iff $stable(\langle x, i\rangle) \supseteq stable(\langle y, j\rangle)$.

Until now we have reduced the problem of filtering the irredundant occurrences of $\mathcal{M}$ to the following problem: Given sets $p_1, \dots, p_m \subseteq \{1, \dots, n\}$, find those p_i such that $p_i \nsubseteq p_j$ for any $j \neq i$. T his problem is called the Maximal Set [246] problem and has received much attention in the past. A more general problem is to find the graph of partial order (called *subset graph*) over a collection of sets. The size of this problem is generally measured by $N = \sum_{i=1}^{m} |p_i|$. Note that in our case $m \in \mathcal{O}(n)$ and so $N \in \mathcal{O}(n^2)$. The size of the subset graph is then $\mathcal{O}(N^2/\log^2 N)$ [191], giving a natural lower bound for any algorithm that computes it. There exists at least one $\mathcal{O}(n * m^2/\log m)$ algorithm for this [81]. Several direct optimisation can be made by pre-calculating an inverse index that gives, for each position the index of sets that contain this position [190]. The case of intersecting two sorted sequences (which is our case), also received attention because of its application to web search engines [24].

We implemented a much simpler algorithm for resolving the special instance of Maximal Set problem, taking advantage that each occurrence has to be compared only to occurrences of motifs that are longer (if not, it cannot be implied), and that the stable position sets can be retrieved already sorted. The total number of code lines for computing the autocorrelation, and filtering the redundant motifs takes approximately just 150 lines, in `C++`. We were able to compute the autocorrelations of DNA sequences of 50,000 base pair on an Intel 2.66 GhZ with 2GB RAM in 5 minutes. For longer sequences, the explicit representation of the autocorrelations (which takes quadratic space) did not fit in the main memory.

On the Use of Irredundant Motifs and Autocorrelations for Compression

As we have seen, the computation of autocorrelations of a sequence is straightforward, while filtering those that are redundant is much more difficult. A natural question is which is the nature of the redundant autocorrelations. In Fig. 5.7 we computed all autocorrelations of the prefix of size 10,000 of `vaccg` (our conclusions holds for any other sequences we analysed) and plotted their lengths against their percentage of don't cares (the opposite of density: a 0 indicates a fully dense motif). Redundant autocorrelations are very few and are mostly between the short ones. On the other hand, irredundant motifs are not very dense: 96% are composed of more than 70% of don't care symbols. For a lossless compressor, this means that special care should be taken when encoding the symbols that disambiguate each don't care.

Even more, most of the autocorrelations occur only twice and this two occurrences do overlap (note that in Fig. (a), all autocorrelations of size bigger than 5,000 trivially overlap). In a LZ77-like parse this could be a minor problem, but for use in a context-free grammar this poses a major hurdle.

Irredundant motifs were used by Apostolico et al. [19] in a LZ78-like parse of the original sequence. The dictionary is set with the motifs chosen by an inexact extension of Greedy, which implies that once the LZ78-like parse selects one motif, all its occurrences have to be replaced. Two kind of results are reported, for lossy compression (the don't cares are not disambiguated) and lossless (an

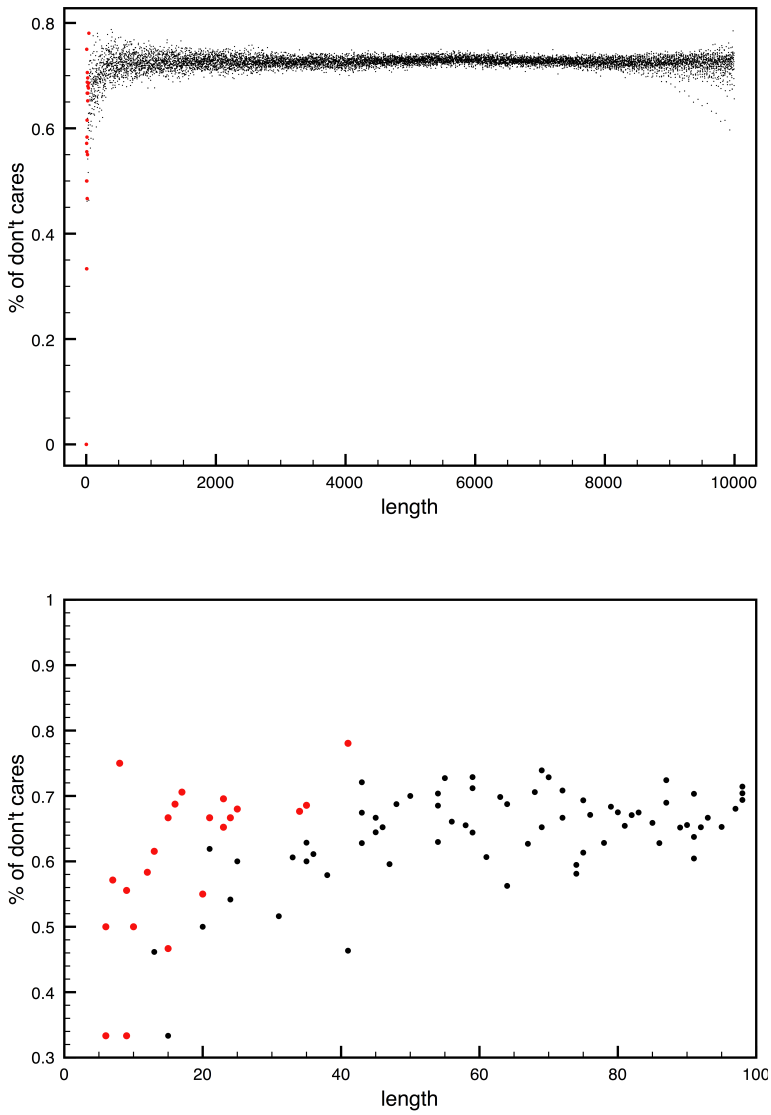


Figure 5.7: Redundant autocorrelations are depicted in red and with a radius three times bigger. (b) is a zoom of (a).

extra stream is added at the end). The GREEDY preprocessing is skipped in [18] and an algorithm is presented that can be considered as the extension of the Ziv-Lempel-Welsh (ZLW) for the case of inexact repeats. For the lossless variant, a set of *resolvers* is added that disambiguate the don't cares. The total number of phrases that belong to the dictionary at the end is reported, and is significantly less than in the case of the traditional ZLW. This algorithm is then improved in Apostolico [12] to a linear variant.

5.4.2 Straight-line Grammars with Don't Cares

In this section we will consider how to extend the notion of straight-line context-free grammars to include rigid patterns. In particular, we would like to keep the uniqueness of the generated sequence, but the presence of "don't care" symbols forces to add information that will specify by which symbols each don't care will be replaced.

We therefore define a **straight-line grammar with don't cares** as a tuple $\langle G, E \rangle$ where G is a straight-line grammar such that $\bullet \in \Sigma$, $E \in \mathcal{N}$ but E does not appear in the derivation of G. We will suppose a total order on the non-terminals, such that S is the maximum element and E the minimum. The right-hand side of E will contain the replacement of the don't care symbols. The derivation proceeds as follows: first, the minimal non-terminal N_1 different from E is replaced everywhere with its corresponding right-hand side α_1. Suppose there are f occurrences of N_1 and α_1 contains d don't care symbols. The first $f \times d$ symbols of the right-hand side of E are then used to replace this don't care symbols, before continuing replacing N_2.

An algorithm that generates a straight-line grammar with don't cares has to face an additional challenge because of the exponential number of rigid patterns. The use of existing maximal classes, which was of great help in the case of exact repeats to get feasible algorithms, does not overcome this difficulty. Regarding maximal repeats, there may still be an exponential number. While irredundant motifs were used successfully for lossy compression schema, their (few) occurrences are prone to overlap and they consist mainly of don't cares. In a lossless context-free grammar-based compressor, where each don't care has to be specified for each occurrence, this may easily become too costly.

In order to limit the number of motifs to consider, we limit here to those that contains exactly a *good* exact repeat. We first select a good exact maximal repeat, and then try to extend it to a maximal non-overlapping motif. E. Ukkonen [234] defines the function $M(w)$ that takes any motif w and computes the only maximal motif of the occurrence-equivalent class of w. Instead of taking the non-overlapping list of motif $M(w)$ we extend w only until it overlaps, and take the list of occurrences inherited from w:

Definition 27 (Motif extension)**.** *Given a motif w over sequence s, $ext(w) = \langle m, o \rangle$, where w appears exactly in m, $o \subseteq pos(m)$ is non-overlapping, $o = \mathcal{L}_s(w) + d$, and m has maximal length. w will be called the* **seed** *of m.*

Inspired by the good performance of IRR-S, we present here a similar algorithm based on rigid patterns. We name this algorithm IMR (for Iterative Motif Replacement) and give pseudocode for it in Algorithm 12. $G_{\langle m,o \rangle \mapsto N}$ is defined as the replacement of the motif m at its occurrence list o ($o \subseteq pos(m)$) by a new symbol N. Also, for each occurrence from left to right, each symbol that is masked by a don't care is appended (from left to right) to the right-hand side of rule E.

From all exact maximal repeat, we select the repeat ω that would reduce the most $\hat{H}(G_{\omega\mapsto N})$. In order to not conflict with the derivation, we enforce that no repeat may contain a don't care symbol. We then extend ω to m using the *ext* function. However, m may present a high number of don't cares. We compute therefore a submotif of m that contains ω: first we find its best extension to the left (line 5) and then to the right.

Algorithm 12 Iterative Motif Replacement (IMR).

Input: s is a sequence
Output: G, a straight-line grammar with don't cares

1: $G \leftarrowtail \langle \Sigma(s), \{S\}, \{S \to s\}, S \rangle$
2: **while** $\exists \omega \ : \ \omega \leftarrowtail \arg\min_{\alpha \in \mathcal{MR}(\mathcal{P})} \hat{H}(G_{\alpha\mapsto N}) \wedge \hat{H}(G_{\alpha\mapsto N}) < \hat{H}(G)$ **do**
3: $\quad m \leftarrowtail ext(\omega)$
4: $\quad j \leftarrowtail$ end position of ω in m
5: $\quad i_\ell \leftarrowtail \arg\min_i \hat{H}(G_{\langle m[i:j], o+i \rangle \mapsto N})$
6: $\quad i_r \leftarrowtail \arg\min_i \hat{H}(G_{\langle m[i_\ell:j+i], o+i_\ell \rangle \mapsto N})$
7: $\quad G \leftarrowtail G_{\langle m[i_\ell:i_r], o+i_\ell \rangle \mapsto N}$
8: **end while**
9: **return** G

Motifs are also searched in the right-hand side of E, but thanks to how don't cares are replaced in the unique derivation, this does not pose problems for the decoder.

The result of IMR^c (which search also in the complement strand) is given in Table 5.7. We achieved our goal of reducing the compression size of those sequences that produced a lot of rules with IRR-S. On the other sequences, the final compression was pretty much the same (note that the exception rule in these cases is empty or very short). Somehow surprising is the better performance of IMR compared to IMR-^{c} on `humghcs`, the most repetitive sequence, We think this may be due to the greedy characteristic of the parsing choice, and that the inclusion of our Minimal Grammar Parsing problem could overcome this difference. Finally, the final compression achieved with the straightforward linear representation and AAC-2 proves to achieve similar ratios as the best DNA compressors.

sequence	IMR			IMR^c		
	bpb	$\|\mathcal{N}\|$	$\|\alpha_E\|$	bpb	$\|\mathcal{N}\|$	$\|\alpha_E\|$
chmpxx	1.8427	16	0	1.6793	25	0
chntxx	1.9379	14	2	1.6196	19	2
hehcmv	1.9649	30	0	1.8542	29	8
humdyst	1.9303	4	0	1.9331	5	0
humghcs	1.1486	248	4,422	1.1820	252	3,635
humhbb	1.8457	37	558	1.8313	44	730
humhdab	1.8763	85	1,435	1.8814	97	397
humprtb	1.8969	72	279	1.8839	77	410
mpomtcg	1.9384	99	289	1.9157	119	443
mtpacga	1.8587	32	74	1.8571	40	76
vaccg	1.8660	27	4	1.7743	18	2

Table 5.7: Compression results of IMR and IMR^c over DNA corpus, number of non-terminals (including S and E) and size of the right-hand side of E.

Chapter 6

Conclusions

Afinal, futebol é bola na rede, o resto é conversa.
BBC Brazil
June, 29th 2009

The real thing in life is what you are doing, why you are searching, and not actually getting the answer. It is like golf: some people think the purpose of golf is getting the ball into the hole, but really the purpose of golf is to have an excuse to be outside. It would be a shame if we actually get to the end.
Donald Knuth
March, 16th 2009

6.1 Summary

Motivated by the problem of deciphering the structure of DNA sequences, we studied the Smallest Grammar Problem. This problem is easy to define and finds numerous uses in a wide range of fields. We identified three big groups of applications, namely Structure Discovery, Kolmogorov Complexity and Data Compression.

Our approach to the general Smallest Grammar Problem was to break it down into two complementary optimisation problems: the choice of constituents, and the choice of which occurrence of these constituents to use in a minimal grammar parsing of these constituents. This decomposition allowed us to present a new formalisation of the Smallest Grammar Problem in form of a complete and correct search space. With respect to the choice of constituents, we analysed the consequences of considering different classes of repeats. Because of the NP-hardness of this problem, we were particularly interested in efficiency issues. Using maximal repeats and overlapping occurrences we reduced the computational complexity of the generic off-line framework IRR from cubic to quadratic. We furthermore accelerated this framework providing an in-place update for the enhanced suffix array used to compute the repeats in each iteration. Regarding the Minimal Grammar Parsing problem, we resolved it in an optimal way with a polynomial algorithm. This enabled us to present different algorithms that outperform the previous best algorithm we had identified, by about 10% in the final grammar size.

Due to the lack of a standard "gold" structure of (non-coding) DNA sequences, we evaluated the quality of the resulting grammars with several approaches. Adopting a Kolmogorov Complexity perspective we evaluated our algorithms using standard experiments showing that they report consistent results. From the Structure Discovery viewpoint, we analysed if, and how, the Smallest Grammar Problem may be useful. In the first part we lower-bounded the worst case for the number of smallest grammars, presenting diverse examples that showed an exponential behaviour. We then analysed and compared minimal grammars on some real-life sequences. We conclude that the huge number of minimal grammars seems to originate from the presence of small constituents. The largest constituents and the highest level of the parse tree remain very stable between minimal grammars.

We put special emphasis on the application of this problem in Data Compression. For this, we studied each step of a grammar-based encoder. In particular, we presented an inference algorithm in the line of the general IRR family that outperforms any other grammar-based DNA compressor. Inspired by the presence of similar repeats in DNA, we then included rigid patterns. These are exact repeats that allow the presence of a "don't care symbol" matching any other symbol. The choice of this special kind of repeat is motivated by the MDL-principle. They allow a very cheap encoding of the mutations or exceptions, while with the edit distance, for instance, we would have to specify the exact position of the changes. Carefully encoding the exceptions allows a lossless recovery of the sequence represented by the grammar. We then implemented an algorithm approximating in each iteration the maximal motif that would compress the most the resulting grammar. Our experiments on the standard corpus yield results in average only less than 5% worse than DNALight [76], the current state of the art. Moreover, considering these inexact repeats allows to obtain a richer structure over the sequence. On one hand, they permit to capture constituents that are not completely identical and specify where the differences (the don't cares) lie. On the other hand the use of rigid patterns allows us to produce a richer parse tree compared to the case of exact repeats were the height of the resulting parse is limited by the size of the longest repeat.

6.2 Perspectives

With respect to our choice of focusing on the *smallest* grammar, our desire of general applicability leaves few choices other than Occam's Razor. During the main part of this thesis, we therefore considered a grammar of minimal size, where size was defined as the number of symbols necessary to represent the grammar. Such a definition does not take into account the size of the alphabet used. Two different grammars with the same size are considered equivalent, without regarding the number of different symbols each one uses. It would be worth considering an MDL-inspired definition of size that also contemplates the growing of the alphabet. The good compression performance of our IRR-S algorithm compared to IRR-MC underlines this point.

Our main motivation to strive for simplicity (and therefore for smallest grammars) was our explicit requirement of not introducing any other structural knowledge as learning bias. The work presented here is just a first step and an ad-hoc application could take advantage of domain knowledge to refrain from the limitations of using size alone as an objective. In the same direction, more information about what the searched structure looks like could be used to define IRR algorithms with more sophisticated score functions. So far, the IRR algorithms presented here consider only length and occurrence number of substrings. Other interesting indicators could be the mutual information of a single word [64] and scores used for automatic keyword detection [113].

Another way of enhancing the final structure is to allow finite recursion, using a recursion counter for instance, but an efficient algorithm for this deserves more research.

We have shown that grammar-based codes have the capacity to compete with other DNA compression algorithms. In particular, the IMR algorithm yields compression rates close to the current best compressors. A direct extension we conceive is to extend the Minimal Grammar Parsing to the case of rigid motifs. A particular characteristic of IMR is that it allows the appearance of *unitarian* rules (of the form $A \rightarrow B$) which are too costly for compression purposes. Combining the MGP solution with a clean-up that removes too costly rules could increase the final compression capacity. It would require some more work to use such an algorithm for efficient compression purpose on big databases. But the main goal of such an algorithm is not necessarily to be a competitive compressor, but to *extract (hierarchical) redundancy.* It should be noted that almost none of the current DNA compressors scale up very well. Kuruppu et al. [137], for instance, report that it takes 93 hours to compress the human genome with *XM* [42], a rather fast algorithm.

A more important characteristic of IMR is that it yields a richer structure. In particular, it is a first step into introducing errors and gaps, a fundamental step to correctly analyse DNA sequences. But the most promising direction we see is to loosen the constraint of the uniqueness of the generated sequence. Our *straight-line grammars with don't cares* are already a step in this direction. If the E rule — which contains the symbols disambiguating the don't cares — is suppressed, then the final grammar could generate more than one sequence. Such a two-step encoding (the model plus the exceptions) lies in the middle between pure straight-line grammars and parse tree compression, two compression frameworks that use formal grammars. In parse tree compressors, both the encoder and the decoder work with the same grammar, and the encoding of the sequence consists in the indexes of the successive production rules to apply. This is applied in cases where a known grammar is available, such as for programming languages [41] or XML documents [111].

Besides generalising the final parse tree, we mentioned a second possible approach to complete an inference process through a learning algorithm that uses the parse tree to provide additional structural information. Another future direction is thus an adaptation of Sakakibara's learning algorithm [202].

With respect to the extension toward a framework consisting of a general model plus the exceptions, Charikar et al. [50] analyse two similar extensions from a theoretical viewpoint. These are *advice grammars* (where a non–straight-line grammar is used, together with an advice string specifying which productions to use during the derivation) and *edit grammars* (where the productions are of the form $A \rightarrow \alpha[e]$, with e a single edit operation). They prove that the size of a smallest such model is equivalent within a constant (logarithmic in the case of edit grammars) factor to the size of a smallest straight-line grammar. Nevertheless, the conclusions of this thesis and the improvement of IMR^c with respect to IRR^c-S allow us to presume that such an approach could make a difference in practical applications.

Similarly Calude et al. [40] use such advice strings in the case of regular automaton. They propose to approximate Kolmogorov Complexity with finite transducers. As the class of finite transducers generates exactly the regular languages, this could therefore be considered as going yet one step lower in the Chomsky hierarchy. In their definition of complexity they use both the size of the topology of the transducer *and* an input string that determines the final string. They also implement an algorithm that computes the finite-state complexity for a string. As expected, this algorithm is unsuitable for application on larger sequences. We collaborated with Tania Roblot to approximate this complexity through the previous computation of a straight-line grammar. Having defined a transformation from a straight-line context-free grammar to a transducer we implemented an algorithm similar to IRRMGP* that optimises the expected size of the final transducer. See Coste and Roblot [73] for details.

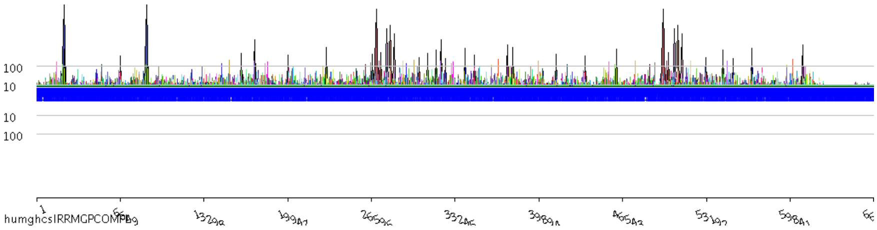
Some of the most successful applications of formal grammars in bioinformatics are based om Stochastic Context-Free Grammars [78, 197, 203]. In a straight-line grammar, the use of probabilities makes no sense, but if the grammar is to be generalised, the use of probabilities could resolve ambiguous parses.

With respect to a further validation of the discovered structure on DNA sequences, we are currently analysing two directions. Michel Termier, Alain Denise and Yann Ponty helped us to design an artificial grammar for a chromosome. Such a definition is hard to achieve, because of varying and unclear definitions of *gene* and alternative splicing, for example. The literature is very sparse on such grammars with non-trivial height, and we report this grammar in Fig. 6.2. This grammar is still a very high-level grammar: the terminals are not nucleotides, but represent known modules. A second direction aims at providing a tool for structure discovery on sequences with an *unknown* structure through a visualisation tool. In particular, the *Pygram* [79] tool was developed to visualise the repeats in form of pyramids over a DNA sequence. Tweaking the input in order to consider only the repeats and occurrences used by a straight-line grammar permits to visualise the parse trees as in Fig. 6.1. The tool also provides an interactive mode to navigate over the sequence.

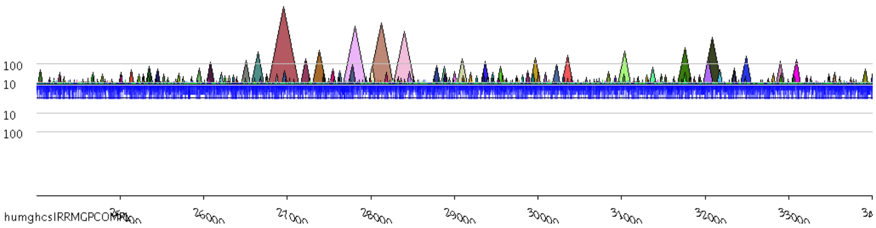
We now come back to our original motivation, namely learning a meaningful structure of one DNA sequence. With respect to model DNA with formal grammars, there seems to be a certain consensus in regarding DNA as a formal language in the most general interpretation. It clearly contains a message and is generated by a yet unknown machinery. It has been long acknowledged that DNA provides examples of non–context-free structures [28, 219]. However, the same is true for natural-language[1], but this did not limit the use of context-free grammars in the first years. Several definitions of Joshi's idea of mildly-context sensitive [120] and other concepts has been given. Joshi himself formalises Tree Adjoining Grammars[2] and examples of other formalisms

[1] Chomsky [59] already suggested that "such grammars are too limited to give a true picture of linguistic structure"

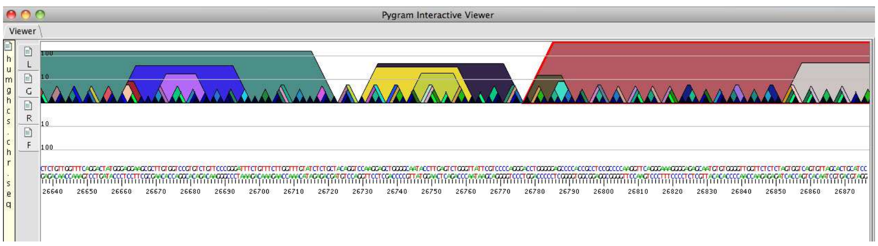
[2] with a dedicated series of conference that in 2010 had its tenth edition



(a) The whole sequence



(b) A zoom



(c) The interactive viewer

Figure 6.1: Some snapshots of the *Pygram* tool visualising the parse tree of `humghcs` obtained with IRRMGP^{c*}.

$$
\begin{array}{lcl}
Chr & \rightarrow & \theta\, s\, \gamma\, s\, \theta \\
x & \rightarrow & \alpha \mid \nu \mid \tau \mid g \\
c & \rightarrow & e\, o\, c \mid e\, æ\, c \mid e \\
æ & \rightarrow & d\, æ'\, \ell\, æ'\, a \\
s & \rightarrow & i\, x\, s \mid i \\
g & \rightarrow & r\, y\, g \mid u\, c\, u \\
o & \rightarrow & d\, o'\, l\, o'\, a
\end{array}
$$

Terminal	Meaning
θ	Telomere
γ	Centromere
i	Intergenic region (IGR)
α	Origin of Replication (ARS)
ν	Non-coding ARN
τ	Transposon
r	Regulatory region
y	Spacer
u	Non-coding but transcript region
e	Exon
d	Donor (transferase process)
ℓ	Lariat
a	Acceptor (transferase process)
o'	Intron
$æ'$	Intron/Exon (alternative splicing)
Non-terminal	**Meaning**
Chr	Chromosome
s	Sequence of genes
x	Gene
g	Coding region + Regulators
c	Coding region for a protein
o	Intron
$æ$	Intron/Exon (alternative splicing)

Figure 6.2: A context-free grammar for a chromosome.

are linear-indexed grammars [97], well-nested multiple context-free grammars (see Kanazawa and Salvati [122] for an overview), re-writing rules [86], dependency models [131], Tree Substitution Grammar (the DOP model, see [32] for an introduction) and Binary Feature Grammars [66, 67]. It seems worth to consider the inference of one derivation, instead of a generic model, in the cases of these richer formalisms. Chiang et al. [57] in particular advocates the use of Tree Adjoining Grammar to model RNA. However, so far there is no consensus about which is a correct formalism. Furthermore, most of these richer models come with an extra cost in the learning process. Given our current state of knowledge of DNA and the difficulty of defining a formal but still accurate model, the use of a context-free grammar (that does capture a lot of typical DNA structures) seems to be a good compromise in a first phase. Moreover, we underline that in this thesis we did not focus on the generative power of these grammars, but rather on the structure they give over the sequence. A context-free grammar excels in this sense thanks to its easy interpretation of the structure in the form of a parse tree.

Appendix A

Corpora

Through this thesis we validate the practical use and compare algorithms on some classical benchmarks. The corpora we use for this are the following:

Canterbury The standard benchmark to evaluate and compare lossless compression methods. For the principle used to select the files belonging to this corpus see Arnold and Bell [23]. Downloaded from http://corpus.canterbury.ac.nz. See Table A.1.

Large Most of the sequences of the Canterbury Corpus are rather short. We will use this corpus when we want to exemplify execution on larger sequences, or have a measure of the growth of the time needed by an algorithm in practice. Downloaded from http://corpus.canterbury.ac.nz/descriptions/#large. See Table A.2.

DNA The standard corpus traditionally used for comparing the performance of DNA compressors. This corpus contains human genes, a chloroplast, some mitochondria and a virus genome. Every sequence contains only four different symbols. It dates from 1993 [106], being the corpus used on the first specific DNA compressor. Of course, the growth of available sequences and of the length of this sequences in particular rises the question a having an up-to-date representative corpus. Downloaded from http://people.unipmn.it/manzini/dnacorpus/historical/. See Table A.3.

sequence	length	$\frac{\#\text{ repeats}}{length}$	$\|\Sigma\|$	description
alice29.txt	152,089	1.45	74	english text (Alice Wonderland)
asyoulik.txt	125,179	1.22	68	english text (Shakespeare, As you Like It)
cp.html	24,603	4.32	86	HTML source (a list of links)
fields.c	11,150	5.03	90	C source code
grammar.lsp	3,721	3.43	76	LISP source code
kennedy.xls	1,029,744	0.08	256	Excel Spreadsheet
lcet10.txt	426,754	2.00	84	english text (technical)
plrabn12.txt	481,861	1.02	81	english text (Milton, Paradise Lost)
ptt5	513,216	194.74	159	fax b/w image
sum	38,240	17.44	255	SPARC executable
xargs.1	4,227	1.77	74	GNU manual page

Table A.1: Description of the Canterbury corpus.

sequence	length	$\frac{\#\text{ repeats}}{length}$	$\lvert\Sigma\rvert$	description
bible.txt	4,047,392	2.57	63	King James version of Bible (english)
e.coli	4,638,690	4.76	4	Complete genome of *E. Coli*
world192.txt	2,473,400	4.47	94	CIA World fact book 1992

Table A.2: Description of the Large Corpus.

sequence	length	$\frac{\#\text{ repeats}}{length}$	$\lvert\Sigma\rvert$	description
chmpxx	121,024	0.82	4	marchantia polymorpha (liverwort) chloroplast
chntxx	155,844	0.77	4	tobacco chloroplast
hehcmv	229,354	1.46	4	human cytomegalovirus (strain AD169)
humdyst	38,770	0.77	4	human dystrophin gene (chr X)
humghcs	66,495	13.77	4	human growth hormone and chorionic somatomammotropin genes (chr 17)
humhbb	73,308	9.01	4	human beta globin region (chr 11)
humhdab	58,864	1.21	4	human contig sequence comprising 3 cosmids (HDAB, HDAC, HDAD)
humprtb	56,737	1.07	4	human hypoxanthine phosphoribosyltransferase (chr X)
mpomtcg	186,609	1.36	4	mitochondria of marchantia polymorpha (liverwort)
mtpacga	100,314	0.97	4	mitochondria of podospora anserina (a filamentous fungus)
vaccg	191,737	2.21	4	vaccinia virus

Table A.3: Description of the DNA corpus.

List of Figures

List of Tables

List of Algorithms

Index

Bibliography

[1] S Abney, S Flickenger, C Gdaniec, C Grishman, P Harrison, D Hindle, R Ingria, F Jelinek, J Klavans, M Liberman, M Marcus, S Roukos, B Santorini, and T Strzalkowski. Procedure for quantitatively comparing the syntactic coverage of english grammars. In E. Black, editor, Workshop on Speech and Natural Language, pages 306–311, Morristown, NJ, USA, 1991. Association for Computational Linguistics. 5.1.2

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms, 2:53–86, February 2004. 3.1, 3.2.2, 3.2.2, 3.4.2

[3] Don Adjeroh, Yong Zhang, Amar Mukherjee, Matt Powell, and Timothy Bell. DNA sequence compression using the Burrows-Wheeler Transform. In IEEE Computer Society Bioinformatics Conference, volume 1, pages 303–13, January 2002. 2.3.3

[4] Pieter Adriaans. Learning as data compression. In Computability in Europe, pages 11–24, Berlin, Heidelberg, 2007. Springer-Verlag. 1.3

[5] Pieter W Adriaans, Marco Vervoort, and P Muidergracht. The EMILE 4.1 grammar induction toolbox. In International Colloquium on Grammatical Inference, January 2002. 1.2.1

[6] Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. Approximating tree edit distance through string edit distance. Algorithmica, 57(2):325–348, January 2010. 5.1.3

[7] Benjamin Allen and Mike Steel. Subtree transfer operations and their induced metrics on evolutionary trees. Annals of combinatorics, 5(1):1–15, 2001. 5.1.3

[8] Lloyd Allison, T Edgoose, and Trevor I Dix. Compression of strings with approximate repeats. In International Conference on Intelligent Systems for Molecular Biology, May 1998. 2.3.3

[9] Miguel Ángel Jiménez-Montaño. On the syntactic structure of protein sequences and the concept of grammar complexity. Bulletin of Mathematical Biology, 46(4):641–659, 1984. 2.4.1, 2.6.7

[10] Natalie Angier. Biologists seek the words in DNA's unbroken text. New York Times, July, 9th 1991. 1.1.1

[11] Dana Angluin. Learning regular sets from queries and counterexamples. Information and Computation, 75(2):87–106, 1987. 1.2, 1.2.2

[12] Alberto Apostolico. Fast gapped variants for Lempel-Ziv-Welch compression. Information and Computation, 205(7):1012–1026, January 2007. 5.4.1

[13] Alberto Apostolico and Stefano Lonardi. Off-line compression by greedy textual substitution. Proceedings of the IEEE, 88:1733–1744, January 2000. 2.6.9, 3.4.1, 4, 5.3.2

[14] Alberto Apostolico and Stefano Lonardi. Compression of biological sequences by greedy off-line textual substitution. In Data Compression Conference, pages 143–153, 2000. 2.6.9, 1, 2.7.3, 4

[15] Alberto Apostolico and Laxmi Parida. Incremental paradigms of motif discovery. Journal Computational Biology, 11(1):15–25, January 2004. 5.4.1

[16] Alberto Apostolico and Franco P Preparata. Data structures and algorithms for the string statistics problem. Algorithmica, 15(5):481–494, January 1996. 3.3

[17] Alberto Apostolico and Claudia Tagliacollo. Optimal offline extraction of irredundant motif bases. In International Computing and Combinatorics Conference, pages 360–371, 2007. 5.4.1

[18] Alberto Apostolico, Matteo Comin, and Laxmi Parida. Motifs in Ziv-Lempel-Welch clef. In Data Compression Conference, pages 1–10, March 2004. 5.4.1

[19] Alberto Apostolico, Matteo Comin, and Laxmi Parida. Bridging lossy and lossless compression by motif pattern discovery. Electronic Notes in Discrete Mathematics, 21:219–225, 2005. 5.4.1

[20] Alberto Apostolico, Matteo Comin, and Laxmi Parida. VARUN: Discovering extensible motifs under saturation constraints. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 99, 2008. 5.4.1

[21] Patrick Argos. The language of protein folding: Many forked tongues. Computers and Chemistry, 16(2):93–102, April 1992. 1.1.2

[22] Hiroki Arimura and Takeaki Uno. An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. Journal of Combinatorial Optimization, 13(3):243–262, 04 2007. 9

[23] Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In Data Compression Conference, pages 201–211, Washington, DC, USA, 1997. IEEE Computer Society. A

[24] Ricardo A Baeza-Yates. A fast set intersection algorithm for sorted sequences. In Combinatorial Pattern Matching, pages 400–408, 2004. 5.4.1

[25] Behshad Behzadi and Fabrice Le Fessant. DNA compression challenge revisited: A dynamic programming approach. In Combinatorial Pattern Matching, 2005. 2.3.3

[26] Timothy Bell, John Cleary, and Ian H Witten. Text Compression. Prentice Hall, 1990. 2.3.1, 2.6.7

[27] Jon Bentley and Douglas McIlroy. Data compression using long common strings. In Data Compression Conference, pages 287–295, March 1999. 2.6.8

[28] Robert C Berwick. The language of genes. In Julio Collado-Vides, Boris Magasanik, and Temple Smith, editors, Integrative approaches to molecular biology. MIT Press, 1996. 6.2

[29] Philip Bille. A survey on tree edit distance and related problems. Theoretical Computer Science, 337(1-3):217–239, January 2005. 5.1.3

[30] Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar compressed strings. In ACM-SIAM Symposium on Discrete Algorithms, January 2011. 2.3.7

[31] Anslem Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. Information Processing Letters, 24:377–380, 1987. 1.3

[32] Rens Bod. The data-oriented parsing approach: Theory and application. In J. Fulcher and L. Jain, editors, The Data-Oriented Parsing Approach: Theory and Application, pages 307–342. Springer, November 2008. 6.2

[33] Sebastian Bonhoeffer, Andras V M Herz, Maarten C Boerlijst, S Nee, Martin A Nowak, and Robert M May. No signs of hidden language in noncoding DNA. Physical Review Letters, 76(11):1977–1977, January 1996. 1.1.2

[34] Sebastian Bonhoeffer, Andreas V M Herz, Maarten C Boerlijst Sean Nee, Martin A Nowak, and Robert M May. Explaining "linguistic features" of noncoding DNA. Science, 271(5245):14–15, 1996. 1.1.2

[35] A Bookstein and ST Klein. Compression, information theory, and grammars: a unified approach. ACM Transactions on Information Systems, 8 (1):27–49, 1990. 5.3.2

[36] Matías Bordese. Análisis y alternativas para la compresión de XML. Master's thesis, FaMAF, Universidad Nacional de Córdoba, Argentina, July 2009. 2.3.6

[37] V Brendel and H G Busse. Genome structure described by formal languages. Nucleic Acids Research, 12(5):2561–2568, 1984. 1.1.3

[38] Gerth Stølting Brodal, Rune Lyngsø, Anna Östlin, and Christian N S Pedersen. Solving the string statistics problem in time O(n log n). In International Colloquium on Automata, Languages, and Programming, pages 728–739, April 2002. 3.3

[39] Frederick Brooks, Jr. Three great challenges for half-century-old computer science. Journal of the ACM, 50(1), January 2003. 1, 2.5

[40] Cristian Calude, Kai Salomaa, and Tania Roblot. Finite-state complexity and the size of transducers. In International Workshop on Descriptional Complexity of Formal Systems, volume 31, pages 38–47, August 2010. 6.2

[41] Robert Cameron. Source encoding using syntactic information source models. IEEE Transactions on Information Theory, 34(4):843–850, July 1988. 6.2

[42] Minh Duc Cao, Trevor I Dix, Lloyd Allison, and Chris Mears. A simple statistical algorithm for biological sequence compression. In Data Compression Conference, 2007. 2.3.3, 6.2

[43] Rafael Carrascosa. Gramáticas Mínimas y descubrimiento de patrones. Master's thesis, Universidad Nacional de Córdoba, February 2010. 4.2.2

[44] Rafael Carrascosa, François Coste, Matthias Gallé, and Gabriel Infante-Lopez. Choosing word occurrences for the smallest grammar problem. In Language and Automata Theory and Applications, February 2010. 4.3

[45] Rafael Carrascosa, François Coste, Matthias Gallé, and Gabriel Infante-Lopez. The smallest grammar problem as constituents choice and minimal grammar parsing. submitted http://www.irisa.fr/symbiose/images/stories/mgalle/papers/sgp_ccmpg.pdf, 2011. 4, 5.1, 5.1.2

[46] Rafael Carrascosa, François Coste, Matthias Gallé, and Gabriel Infante-Lopez. Searching for smallest grammars on large sequences and application to DNA. Journal of Discrete Algorithms, 2011. 3, 4

[47] Daniele Cerra and Mihai Datcu. A similarity measure using smallest context-free grammars. In Data Compression Conference, pages 346–355, 2010. 2.4.2

[48] Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. ACM Transactions on Algorithms, 3(2), May 2007. 3.4.1

[49] CH Chang. DNAC: A compression algorithm for DNA sequences by nonoverlapping approximate repeats. Master's thesis, National Taiwan University, Taiwan, 2004. 2.3.3

[50] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and Abhi Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In Annual ACM Symposium on Theory of Computing, January 2002. 2.2, 2.4.1, 6.2

[51] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and abhi shelat. The smallest grammar problem. IEEE Transactions on Information Theory, 51(7):2554–2576, July 2005. 1, 2.2, 2.3.4, 2.6.1, 2.6.5, 2.6.11, 2.7, 3.2.3, 4, 1, 5.3

[52] CA Chatzidimitriou-Dreismann, RMF Streffer, and D Larhammar. Lack of biological significance in the 'linguistic features' of noncoding DNA – a quantitative analysis. Nucleic Acids Research, 24(9):1676–1681, January 1996. 1.1.2

[53] Gang Chen, Simon Puglisi, and William F Smyth. Lempel-Ziv factorization using less time & space. Mathematics in Computer Science, January 2008. 2.6.1

[54] Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for DNA sequences. Engineering in Medicine and Biology Magazine, January 2001. 2.3.3

[55] Xin Chen, Ming Li, Bin Ma, and John Tromp. DNACompress: fast and effective DNA sequence compression. Bioinformatics, January 2002. 2.3.3, 5.4

[56] Neva Cherniavsky and Richard Lander. Grammar-based compression of DNA sequences. In DIMACS Working Group on The Burrows-Wheeler Transform, page 21, 2004. 2.3.4, 2.6.4, 3, 5.3

[57] David Chiang, Aravind Joshi, and David B Searls. Grammatical representations of macromolecular structure. Journal of Computational Biology, 13(5):1077–1100, January 2006. 1.1.3, 6.2

[58] Sheng-Yuan Chiu, Wing-Kai Hon, Rahul Shah, and Jeffrey Vitter. I/O-efficient compressed text indexes: From theory to practice. In Data Compression Conference, pages 426–434, 2010. 2.3.7

[59] Noam Chomsky. Three models for the description of language. IEEE Transactions on Information Theory, 2(3):113–124, January 1956. 1, 1.2, 1

[60] Noam Chomsky. Syntactic Structures. Mouton and Co., 1957. 1.2

[61] Scott Christley, Yiming Lu, Chen Li, and Xiaohui Xie. Human genomes as email attachments. Bioinformatics, 25(2):274–275, January 2009. 2.3.3

[62] Rudi Cilibrasi. Statistical Inference Through Data Compression. PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, December 2007. 2.4.2

[63] Rudi Cilibrasi and Paul Vitanyi. Clustering by compression. IEEE Transactions on Information Theory, 51(4):1523–1545, 2005. 2.4.2, 5.2, 5.2.2

[64] Alexander Clark. Learning deterministic context free grammars: The Omphalos competition. Machine Learning, pages 93–110, January 2007. 1.2.1, 6.2

[65] Alexander Clark. Three learnable models for the description of language. In Language and Automata Theory and Applications, pages 16–31, 2010. 1.1.3

[66] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A polynomial algorithm for the inference of context free languages. In International Colloquium on Grammatical Inference, July 2008. 1.2.1, 6.2

[67] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A note on contextual binary feature grammars. In EACL Workshop on Computational Linguistic Aspects of Grammatical Inference, February 2009. 6.2

[68] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. Fundamenta Informaticae, August 2010. 2.3.7

[69] Francisco Claude, Antonio Fari na, Miguel Martínez-Prieto, and Gonzalo Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In International Conference on Bioinformatics and Bioengineering, 2010. 2.3.7

[70] Julio Collado-Vides. The search for a grammatical theory of gene regulation is formally justified by showing the inadequacy of context-free grammars. Bioinformatics, 7(3):321, 1991. 1.1.3

[71] Julio Collado-Vides. Grammatical model of the regulation of gene expression. Proceedings of the National Academy of Sciences, 89(20):9405–9409, January 1992. 1.1.3

[72] Matteo Comin and Davide Verzotto. Classification of protein sequences by means of irredundant patterns. BMC Bioinformatics, 11, 2010. 5.4.1

[73] François Coste and Tania K Roblot. Towards evaluating the finite-state complexity of DNA. Technical report, INRIA Rennes – Bretagne Atlantique, 2010. 6.2

[74] Francis Crick. Central dogma of molecular biology. Nature, 227(5258): 561–563, 1970. 1.1.1

[75] Colin de la Higuera. Grammatical Inference Learning Automata and Grammars. Cambridge University Press,, 2010. 1.2

[76] Sérgio Deusdado. Análise e compressão de sequências genómicas. PhD thesis, Universidade do Minho, Portugal, July 2008. 2.3.3, 6.1

[77] Pedro Domingos. The role of Occam's Razor in knowledge discovery. Data Mining and Knowledge Discovery, January 1999. 1.3

[78] Robin D Dowell and Sean R Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. BMC Bioinformatics, page 14, June 2004. 6.2

[79] Patrick Durand, F Mahé, A Valin, and Francois Nicolas. Browsing repeats in genomes: Pygram and an application to non-coding region analysis. BMC Bioinformatics, January 2006. 6.2

[80] Werner Ebeling and Miguel A Jiménez-Montaño. On grammars, complexity, and information measures of biological macromolecules. Mathematical Biosciences, 52(1–2):53–71, November 1980. 2.2, 2.4.1

[81] Amr Elmasry. The subset partial order: Computing and combinatorics. In Workshop on Analytic Algorithmics and Combinatorics, 2010. 5.4.1

[82] Scott Charles Evans. Kolmogorov complexity estimation and application for information system security. PhD thesis, Rensselaer Polytechnic Institute, August 2003. 2.5, 2.6.10, 2.7.2

[83] Scott Charles Evans, Bruce Barnett, Stephen Bush, and Gary J Saulnier. Minimum description length principles for detection and classification of FTP exploits. In IEEE Military Communications Conference, volume 1, pages 473–479, January 2004. 2.6.10

[84] Scott Charles Evans, Antonis Kourtidis, T Stephen Markham, Jonathan Miller, Douglas S Conklin, and Andrew S Torres. MicroRNA target detection and analysis for genes related to breast cancer using MDLcompress. EURASIP Journal on Bioinformatics and Systems Biology, 2007. 2.5, 2.6.10, 4, 2.7.3, 4

[85] Rémi Eyraud. Inférence Grammaticale de Langages Hors-Contextes. PhD thesis, Université Jean Monnet de Saint-Étienne, February 2006. 1.2.2

[86] Rémi Eyraud, Colin de la Higuera, and Jean-Christophe Janodet. LARS: A learning algorithm for rewriting systems. Machine Learning, January 2007. 6.2

[87] Paolo Ferragina. Data structures: Time, I/Os, entropy, joules! invited talk European Symposium on Algorithms, 2010. 2.3.7

[88] Paolo Ferragina, Roberto Grossi, and Manuela Montangero. On updating suffix tree labels. Theoretical Computer Science, 201(1-2):249–262, 1998. 3.4.1

[89] Paolo Ferragina, Raffaele Giancarlo, Valentina Greco, Giovanni Manzini, and Gabriel Valiente. Compression-based classification of biological sequences and structures via the universal similarity metric: experimental assessment. BMC Bioinformatics, 8:252, Jan 2007. 2.4.2, 5.2, 5.2.1, 2, 5.4, A

[90] Edward Fiala and Daniel H Greene. Data compression with finite windows. Communications of the ACM, 32(4):490–505, 1989. 3.4.1

[91] F Flam. Hints of a language in junk DNA. Science, 266(5189):1320, 1994. 1.1.2

[92] Philip Gage. A new algorithm for data compression. The C Users Journal, 12(2), February 1994. 2.6.7

[93] Travis Gagie and Paweł Gawrychowski. Grammar-based compression in a streaming model. In Language and Automata Theory and Applications, pages 273–284, 2010. 2.6.11

[94] Matthias Gallé. A new tree distance metric for structural comparison of sequences. In Alberto Apostolico, Andreas Dress, and Laxmi Parida, editors, Structure Discovery in Biology: Motifs, Networks & Phylogenies, number 10231 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. 5.1.3

[95] Matthias Gallé, Pierre Peterlongo, and François Coste. In-place update of suffix array while recoding words. In Jan Holub and Jan Žďárek, editors, Prague Stringology Conference, pages 54–67, Czech Technical University in Prague, Czech Republic, 2008. 3

[96] Matthias Gallé, Pierre Peterlongo, and François Coste. In-place update of suffix array while recoding words. International Journal of Foundations of Computer Science, 20(6):1025–1045, January 2009. 3

[97] Gerald Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, Natural Language Parsing and Linguistic Theories, pages 69–94. Springer, 1988. 6.2

[98] Raffaele Giancarlo, Davide Scaturro, and Filippo Utro. Textual data compression in computational biology: a synopsis. Bioinformatics, 25(13): 1575–86, July 2009. 2.3.3

[99] Mario Gimona. Protein linguistics - a grammar for modular protein assembly? Nature Reviews Molecular Cell Biology, 7(1):68–73, January 2006. 1.1.3

[100] Leszek Gąsieniec, Roman Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In Data Compression Conference, 2005. 2.3.7

[101] Özkan U Nalbanto glu, David J Russell, and Khalid Sayood. Data compression concepts and algorithms and their applications to bioinformatics. Entropy, 12(1):34–52, January 2010. 2.3.3

[102] E Mark Gold. Language identification in the limit. Information and Control, 10(5):447–474, 1967. 1.2

[103] E Mark Gold. Complexity of automaton identification from given data. Information and Control, 37(3):302–320, 1978. 1.2

[104] Michael Gribskov. The language metaphor in sequence analysis. Computers and Chemistry, 16(2):85–88, April 1992. 1.1.2

[105] Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, and Fabio Vandin. MADMX: A novel strategy for maximal dense motif extraction. In Workshop on Algorithms in Bioinformatics, pages 362–374, 2009. 5.4.1

[106] Stéphane Grumbach and Fariza Tahi. A new challenge for compression algorithms: Genetic sequences. In Data Compression Conference, 1993. 2.3.3, A

[107] Stéphane Grumbach and Fariza Tahi. A new challenge for compression algorithms: Genetic sequences. Information Processing and Management, 30(6):875–886, 1994. 2.3.3, 2.7.3, 5.3.2

[108] Peter Grünwald. The Minimum Description Length principle. MIT Press, 2007. 1.3

[109] Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In ACM-SIAM symposium on Discrete Algorithms, pages 697–704, 1994. 3.4.1

[110] Dan Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, January 1997. 3.2, 3.2, 3.2.1, 3.2.2, 3.2.2

[111] S Harrusi, A Averbuch, and A Yehudai. XML syntax conscious compression. In Data Compression Conference, 2006. 6.2

[112] Danny Hermelin, Gad M Landau, Shir Landau, and Orenw Weimann. Unified compression-based acceleration of edit-distance computation. Arxiv preprint arXiv:1004.1194, 2010. 2.3.7

[113] Juan Herrera and Pedro Pury. Statistical keyword detection in literary corpora. European Physical Journal, May 2008. 6.2

[114] Wing-Kai Hon, Rahul Shah, and Jeffrey S Vitter. Compression, indexing, and retrieval for massive string data. invited talk Combinatorial Pattern Matching, 2010. 2.3.7

[115] John E Hopcroft and Jefrey D Ullman. Introduction to Automata Theory, Languages and Computation. Adison-Wesley Publishing Company, 1979. 2.1.2

[116] Shunsuke Inenaga, Takashi Funamoto, Masayuki Takeda, and Ayumi Shinohara. Linear-time off-line text compression by longest-first substitution. In String Processing and Information Retrieval, volume 2857, pages 137–152, 2003. 2.6.8

[117] Nathan E Israeloff, M Kagalenko, and K Chan. Can Zipf distinguish language from noise in noncoding DNA? Physical Review Letters, 76: 1976–1976, 1996. 1.1.2

[118] Sungchul Ji. The linguistics of DNA: words, sentences, grammar, phonetics, and semantics. Annals of the New York Academy of Sciences, 870: 411–417, 1999. 1.1.2

[119] Sungchul Ji. The cell as the smallest DNA-based molecular computer. Biosystems, 52(1-3):123–133, January 1999. 1.1.2

[120] Aravind Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In Natural Language Parsing, Psycological, Computational and Theoretical Perspectives. Cambridge University Press, May 1985. 6.2

[121] Horace Freeland Judson. The Eighth Day of Creation: Makers of the Revolution in Biology. Simon and Schuster, 1979. 1.1.1

[122] Makoto Kanazawa and Sylvain Salvati. The copying power of well-nested multiple context-free grammars. In Language and Automata Theory and Applications, January 2010. 6.2

[123] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In International Conference on Automata, Languages and Programming, pages 943–955. Springer, 2003. 3.1, 3.4.4

[124] Jyrki Katajainen and Timo Raita. An analysis of the longest match and the greedy heuristics in text encoding. Journal of the ACM, 39(2), April 1992. 2.3.1

[125] Goulven Kerbellec. Apprentissage d'automates modélisant des familles de séquences protéiques. PhD thesis, Université de Rennes 1, April 2008. 1

[126] Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Collage system: a unifying framework for compressed pattern matching. Theoretical Computer Science, 298(1):253–272, January 2003. 2.3.7

[127] John C Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. IEEE Transactions on Information Theory, 46(3):737–754, May 2000. 2.2, 2.3.4, 4, 2.6.5, 2.6.6, 3.2.3

[128] John C Kieffer, En-Hui Yang, Gregory Nelson, and Pamela Cosman. Universal lossless compression via multilevel pattern matching. IEEE Transactions on Information Theory, 46:1227–1245, January 2000. 2.6.6

[129] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear time construction of suffix arrays. In Combinatorial Pattern Matching, pages 186–2003, 2003. 3.1

[130] Dan Klein. The Unsupervised Learning of Natural Language Structure. PhD thesis, University of Stanford, 2005. 5.1.2

[131] Dan Klein and Christopher D Manning. Corpus-based induction of syntactic structure: Models of dependency and constituency. In Association for Computational Linguistics, January 2004. 6.2

[132] Friedhart Klix. Struktur, Strukturbeschreibung und Erkennungsleistung. In Friedhart Klix, editor, Organismische Informationsverarbeitung: Zeichenerkennung, Begriffsbildung, Problemlösen, pages 110–130, 108 Berlin, Leipziger Str.3–4, 1973. Akademie-Verlag. 2.4.1

[133] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Combinatorial Pattern Matching, pages 200–210. Springer, 2003. 3.1

[134] Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In Symposium on Foundations of Computer Science, pages 596–604, New York, USA, 1999. IEEE. 3.2.2

[135] Gergely Korodi and Ioan Tabus. Compression of annotated nucleotide sequences. IEEE/ACM Transactions on Computational Biology and Bioinformatics, January 2007. 2.3.3

[136] Harold W Kuhn. The Hungarian method for the assignment problem. Naval Research Logistics Quarterly, 2:83–87, 1955. 5.1.3

[137] Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, and Justin Zobel. Repetition-based compression of large DNA datasets. In International Conference on Computational Molecular Biology, April 2009. 2.3.3, 6.2

[138] J Kevin Lanctot, M Li, and En hui Yang. Estimating DNA sequence entropy. In ACM-SIAM Symposium on Discrete Algorithms, pages 409–418, January 2000. 2.5, 2.6.8, 2, 2.7.3, 3.4.1

[139] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. Proceedings of the IEEE, 88(11):1722–1732, November 2000. 2.6.7, 3

[140] N Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical report, Department of Computer Science, Lund University, Sweden, May 1999. 3.1, 3.4.4

[141] Eric Lehman. Approximation algorithms for grammar-based data compression. Master's thesis, Massachusetts Institute of Technology, 2002. 2.2, 2.6.11

[142] G Leighton, J Diamond, and T Muldner. AXECHOP: a grammar-based compressor for XML. In Data Compression Conference, page 467, March 2005. uses Multilevel Pattern Matching from KY. 2.3.6

[143] G Leighton, James Diamond, and T Müldner. A grammar-based approach for compressing XML. Technical report, Acadia University, August 2005. 2.3.6

[144] Siu-Wai Leung, Chris Mellish, and Dave Robertson. Basic gene grammars and DNA-ChartParser for language processing of escherichia coli promoter DNA sequences. Bioinformatics, 17(3):226–236, January 2001. 1.1.3

[145] Ming Li and Paul Vitányi. An Introduction to Kolmogorov Complexity and its Applications. Springer Verlag, third edition, 2008. 2.4

[146] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul Vitányi. The similarity metric. IEEE Transactions on Information Theory, 50(12):3250–3264, March 2003. 2.4.2

[147] Yongjing Lin, Youtao Zhang, Quanzhong Li, and Jun Yang. Supporting efficient query processing on compressed XML files. In ACM symposium on Applied computing, pages 660–665, 2005. 2.3.6

[148] Alan H Lipkus. A proof of the triangle inequality for the Tanimoto distance. Journal Mathematical Chemistry, 26(1-3):263–265, January 1999. 5.1.3

[149] Qi Liu, Yu Yang, Chun Chen, Jiajun Bu, Yin Zhang, and Xiuzi Ye. RNACompress: Grammar-based compression and informational complexity measurement of RNA secondary structure. BMC Bioinformatics, 9:176, January 2008. 2.3.5, 5.3

[150] David Loewenstern and Peter N Yianilos. Significantly lower entropy estimates for natural DNA sequences. Journal of Computational Biology, 6, February 1999. 2.3.3

[151] Christopher Loose, Kyle Jensen, Isidore Rigoutsos, and Gregory Stephanopoulos. A linguistic model for the rational design of antimicrobial peptides. Nature, 443(7113):867–869, January 2006. 1.1.3

[152] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. Bioinformatics, 18(3):440–5, March 2002. 2.3.3

[153] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In ACM-SIAM Symposium on Discrete Algorithms, pages 319–327, 1991. 3.1

[154] Michael A Maniscalco and Simon J Puglisi. An efficient, versatile approach to suffix sorting. ACM Journal of Experimental Algorithmics, 12:1–23, 2008. 2

[155] R N Mantegna, S V Buldyrev, A L Goldberger, S Havlin, C K Peng, M Simons, and H E Stanley. Linguistic features of noncoding DNA sequences. Physical Review Letters, 73(23):3169–3172, 1994. 1.1.2

[156] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. Algorithmica, 40(1):33–50, 2004. 3.1, 1

[157] Giovanni Manzini and Marcella Rastero. A simple and fast DNA compressor. Software - Practice and Experience, February 2004. 2.3.3

[158] T Stephen Markham, Scott C Evans, Jeremy Impson, and Eric Steinbrecher. Implementation of an incremental MDL-based two part compression algorithm for model inference. In Data Compression Conference, pages 322–331, 2009. 3, 5.3

[159] Shirou Maruyama, Hiromitsu Miyagawa, and Hiroshi Sakamoto. Improving time and space complexity for compressed pattern matching. In International Symposium Algorithms and Computation, pages 484–493, 2006. 2.3.7

[160] Shirou Maruyama, Yohei Tanaka, Hiroshi Sakamoto, and Masayuki Takeda. Context-sensitive grammar transform: Compression and pattern matching. In String Processing and Information Retrieval, January 2008. 2.3.7

[161] Toshiko Matsumoto, Kunihiko Sadakane, and Hiroshi Imai. Biological sequence compression algorithms. In Genome informatics Workshop on Genome Informatics, volume 11, pages 43–52, January 2000. 2.3.3

[162] Edward M McCreight. A space-economical suffix tree construction algorithm. Journal ACM, 23(2):262–272, 1976. 3.4.1

[163] David McKay. Information Theory, Inference, and Learning Algorithms. Cambridge University Press, 2003. 2.3

[164] Aleksandar Milosavljević and Jerzy Jurka. Discovery by minimal length encoding: A case study in molecular evolution. Machine Learning, 12(1): 69–87, 1993. 3, 2.3.3

[165] Yuta Mori. libdivsufsort project (libdivsufsort-2.0.0), August 2008. `http://code.google.com/p/libdivsufsort/`. 1, 3.4.4

[166] Yuta Mori. An implementation of the induced sorting algorithm, 2008. `http://yuta.256.googlepages.com/sais`. 3.4.4

[167] Ryosuke Nakamura, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Simple linear-time off-line text compression by longest-first substitution. In Data Compression Conference, pages 123–132, Washington, DC, USA, 2007. IEEE Computer Society. 2.6.8

[168] Ryosuke Nakamura, Shunsuke Inenaga, Hideo Bannai, Takashi Funamoto, Masayuki Takeda, and Ayumi Shinohara. Linear-time text compression by longest-first substitution. Algorithms, 2(4):1429–1448, 2009. 2.6.8, 12, 3, 3.4.1

[169] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. ACM Computing Surveys, 39(1):2, 2007. 2.3.7

[170] Gonzalo Navarro and Luís Russo. Re-pair achieves high-order entropy. In Data Compression Conference, page 537, 2008. 2.6.7

[171] Craig G Nevill-Manning. Inferring Sequential Structure. PhD thesis, University of Waikato, 1996. 1.2.1, 2.5, 2.3, 5.1

[172] Craig G Nevill-Manning and Ian H Witten. Compression and explanation using hierarchical grammars. The Computer Journal, 40(2,3):103–116, February 1997. 2.2, 2.3, 5, 5.1

[173] Craig G Nevill-Manning and Ian H Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. Journal of Artificial Intelligence Research, 7, January 1997. 2.3, 5.1

[174] Craig G Nevill-Manning and Ian H Witten. Linear-time, incremental hierarchy inference for compression. In Data Compression Conference, 1997. 2.3

[175] Craig G Nevill-Manning and Ian H Witten. Inferring lexical and grammatical structure from sequences. In Compression and Complexity of Sequences, 1997. 2.3

[176] Craig G Nevill-Manning and Ian H Witten. Phrase hierarchy inference and compression in bounded space. In Data Compression Conference, pages 179–188, 1998. 2.3

[177] Craig G Nevill-Manning and Ian H Witten. On-line and off-line heuristics for inferring hierarchies of repetitions in sequences. In Data Compression Conference, pages 1745–1755. IEEE, November 2000. 2.3, 2.6.8, 2.6.9, 2.7, 3.4.1

[178] Craig G Nevill-Manning, Ian H Witten, and David Maulsby. Compression by induction of hierarchical grammars. In Data Compression Conference, pages 244–253, 1994. 2.3.4, 2.3, 4, 5.3

[179] Craig G Nevill-Manning, Ian H Witten, and Dan Olsen, Jr. Compressing semi-structured text using hierarchical phrase identification. In Data Compression Conference, January 1996. 2.3

[180] Craig G Nevill-Manning, Ian H Witten, and Gordon W Paynter. Browsing in digital libraries: a phrase-based approach. In International Conference on Digital libraries, pages 230–236, 1997. 2.3

[181] Craig G Nevill-Manning, Ian H Witten, and Gordon W Paynter. Lexically-generated subject hierarchies for browsing large collection. International Journal of Digital Libraries, 2/3(111–123), 1999. 2.3

[182] Jacques Nicolas, Patrick Durand, Grégory Ranchy, Sébastien Tempel, and AS Valin. Suffix-tree analyser (STAN): looking for nucleotidic and peptidic patterns in chromosomes. Bioinformatics, 21(24):4408–4410, January 2005. 1.1.3

[183] Jacques Nicolas, Christine Rousseau, Anne Siegel, Pierre Siegel, François Coste, Patrick Durand, Sébastien Tempel, Anne-Sophie Valin, and Frédéric Mahé. Modeling local repeats on genomic sequences. Technical report, INRIA, 2008. 3.2

[184] Laxmi Parida, Isidore Rigoutsos, Aris Floratos, Dan Platt, and Yuan Gao. Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and polynomial time algorithms. In ACM-SIAM Symposium on Discrete Algorithms, pages 297–308, 2000. 5.4.1

[185] Johann Pelfrêne, Saïd Abdeddaïm, and Joël Alexandre. Extracting approximate patterns. Journal of Discrete Algorithms, 3(2-4):293–320, 2005. 8, 9, 5.4.1

[186] Matthieu Perrin. Compression de séquences d'ADN à base de grammaires minimales. Technical report, ENS Cachan/Bretagne, 2010. 5.3, 5.3.1, 5.3.1

[187] Nadia Pisanti, Maxime Crochemore, Roberto Grossi, and Marie-France Sagot. A comparative study of bases for motif inference. In String Algorithmics, pages 195–225. KCL Press, 2004. 5.4.1

[188] Nadia Pisanti, Maxime Crochemore, Roberto Grossi, and Marie-France Sagot. Bases of motifs for generating repeated patterns with wild cards. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2(1), 2005. 5.4.1, 5.4.1

[189] O Popov, DM Segal, and Edward N Trifonov. Linguistic complexity of protein sequences as compared to texts of human languages. Biosystems, 38(1):65–74, January 1996. 1.1.2

[190] Paul Pritchard. A simple sub-quadratic algorithm for computing the subset partial order. Information Processing Letters, 56(6):337–341, 1995. 5.4.1

[191] Paul Pritchard. On computing the subset graph of a collection of sets. Journal of Algorithms, 33(2):187–203, 1999. 5.4.1

[192] Simon Puglisi, William F Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. ACM Computing Surveys, 39(2), July 2007. 3.1, 3.4.4

[193] Simon J Puglisi, William F Smyth, and Munina Yusufu. Fast optimal algorithms for computing all the repeats in a string. In Jan Holub and Jan Zdarek, editors, Prague Stringology Conference, pages 161–169, 2008. 3.2.2, 3.2.2

[194] I Rigoutsos, T Huynh, K Miranda, A Tsirigos, A McHardy, and D Platt. Short blocks from the noncoding parts of the human genome have instances within nearly all known genes and relate to biological processes. PNAS, 2006. 1.3

[195] Eric Rivals, Jean-Paul Delahaye, Olivier Delgrange, and Max Dauchet. A guaranteed compression scheme for repetitive DNA sequences. In Data Compression Conference, January 1996. 2.3.3

[196] Eric Rivals, Olivier Delgrange, Jean-Paul Delahaye, and Max Dauchet. Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences. Bioinformatics, January 1997. 1.3, 2.3.3

[197] E Rivas and SR Eddy. The language of RNA: a formal grammar that includes pseudoknots. Bioinformatics, 16(4):334, 2000. 6.2

[198] Judith Roof. The poetics of DNA. U of Minnesota Press, 2007. 1.1.1

[199] Christine Rousseau, Mathieu Gonnet, Marc Le Romancer, and Jacques Nicolas. CRISPI: a CRISPR interactive database. Bioinformatics, 25(24), 2009. 3.2

[200] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science, 302(1-3):211–222, 2003. 2.6.1, 2.6.11, 2.7.2

[201] Süleyman Cenk Sahinalp and U Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In Symposium on Foundations of Computer Science, page 320, Washington, DC, USA, 1996. IEEE Computer Society. 3.4.1

[202] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. Information and Computation, 97(1):23–60, 1992. 1.2, 1, 6.2

[203] Yasubumi Sakakibara. Learning context-free grammars using tabular representations. Pattern Recognition, 38(9):1372–1383, December 2005. 6.2

[204] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. In Combinatorial Pattern Matching, volume 2676, pages 348–360, January 2003. 2.6.11

[205] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. Journal of Discrete Algorithms, 3(2–4):416–430, January 2005. 2.6.11

[206] Hiroshi Sakamoto, Takuya Kida, and Shinichi Shimozono. A space-saving linear-time algorithm for grammar-based compression. In String Processing and Information Retrieval, pages 218–229, 2004. 2.6.11

[207] Sherif Sakr. XML compression techniques: A survey and comparison. Journal of Computer and System Sciences, 75(5):303–322, January 2009. 2.3.6

[208] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic Burrows-Wheeler transform. In Jan Holub and Jan Žďárek, editors, Prague Stringology Conference, pages 13–25, Czech Technical University in Prague, Czech Republic, 2008. 3.4.1

[209] Geoffrey Sampson. A proposal for improving the measurement of parse accuracy. International Journal of Corpus Linguistics, 5:53–68, 2000. 5.1.2

[210] Hisahiko Sato, Takashi Yoshioka, Akihiko Konagaya, and Tetsuro Toyoda. DNA data compression in the post genome era. Genome Informatics Series, pages 512–514, 2001. 2.3.3

[211] Ulrich Scheidereiter. Zur Beschreibung strukturierter Objeckte mit kontextfreien Grammatiken. In Friedhart Klix, editor, Organismische Informationsverarbeitung: Zeichenerkennung, Begriffsbildung, Problemlösen, pages 131–135, 108 Berlin, Leipziger Str.3–4, 1973. Akademie-Verlag. 1, 2.2, 2.4.1, 5.1

[212] Michael Schrage. Learning to speak the language of life. Los Angeles Times, April, 22nd 1993. 1.1.1

[213] Ernst J Schuegraf and H S Heaps. A comparison of algorithms for data base compression by use of fragments as language elements. Information Storage and Retrieval, 10:309–319, 1974. 2.3.1, 4.1.1

[214] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. Software - Practice and Experience, 37(3): 309–329, 2007. 3.4.4

[215] David B Searls. The computational linguistics of biological sequences. In Lawrence Hunter, editor, Artificial Intelligence and Molecular Biology, page 75. AAAI Press Copublications, March 1993. 1.1.3, A

[216] David B Searls. String variable grammar: A logic grammar formalism for the biological language of DNA. Journal of Logic Programming, 24(1-2): 73–102, January 1995. 1.1.3

[217] David B Searls. Linguistic approaches to biological sequences. Computer Applications in the Biosciences, 13(4):333–344, January 1997. 1.1.3

[218] David B Searls. Reading the book of life. Bioinformatics, January 2001. 1.1.3

[219] David B Searls. The language of genes. Nature, February 2002. 1, 1.1.2, 1.1.3, 6.2

[220] David B Searls. Linguistics: trees of life and of language. Nature, 426 (6965):391–2, November 2003. 1.1.3

[221] Abhi Shelat. Evaluating grammar-based data compression algorithms. Master's thesis, Massachusetts Institute of Technology, 2001. 4

[222] Jeong Seop Sim. Time and space efficient search for small alphabets with suffix arrays. In Conference on Fuzzy Systems and Knowledge Discovery, pages 1102–1107, 2005. 3.4.2

[223] Matthew Simon. Emergent Computation: Emphasizing Bioinformatics. Springer, 2005. 1.1.3

[224] Zach Solan, David Horn, Eytan Ruppin, and Shimon Edelman. Unsupervised learning of natural languages. Proceedings of the National Academy of Sciences, January 2005. 1.2.1

[225] James A Storer and Thomas G Szymanski. Data compression via textual substitution. Journal of the ACM, 29(4):928–951, April 1982. 1, 2.2, 2.3.1

[226] Bonnie J Strait and T Gregory Dewey. The Shannon information entropy of protein sequences. Biophysical Journal, 71(1):148–155, January 1996. 1.1.2

[227] Ioan Tabus, Gergely Korodi, and Jorma Rissanen. DNA sequence compression using the normalized maximum likelihood model for discrete regression. In Data Compression Conference, page 10, February 2003. 2.3.3

[228] Vojtech Toman. Compression of XML data. In International Conference on Advanced Information Systems Engineering, pages 1–12, April 2004. 2.3.6

[229] Helene Touzet. A linear tree edit distance algorithm for similar ordered trees. In Combinatorial Pattern Matching, volume 3537, pages 334–345, January 2005. 5.1.3

[230] Edward N Trifonov. The multiple codes of nucleotide sequences. Bulletin of Mathematical Biology, 51(4):417–432, 1989. 1.1.2

[231] Edward N Trifonov and Volker Brendel. Gnomic: a dictionary of genetic codes. Balaban, 1986. 1.1.2

[232] Anastasios A Tsonis, James B Elsner, and Panagiotis A Tsonis. Is DNA a language? Journal of Theoretical Biology, 184(1):25–29, 1997. 1.1.2

[233] Esko Ukkonen. On-line construction of suffix trees. Algorithmica, 14: 249–260, 1995. 3.4.2

[234] Esko Ukkonen. Maximal and minimal representations of gapped and non-gapped motifs of a string. Theoretical Computer Science, 410(43): 4341–4349, 2009. String Algorithmics: Dedicated to Professor Maxime Crochemore on the occasion of his 60th birthday. 5.4.1, 5.4.2

[235] Menno van Zaanen. ABL: Alignment-Based Learning. In International Conference on Computational Linguistics, 2000. 1.2.1

[236] Menno van Zaanen. Bootstrapping Structure into Language: Alignment-Based Learning. PhD thesis, University of Leeds, February 2002. 1.2.1

[237] Menno van Zaanen and Pieter W Adriaans. Comparing two unsupervised grammar induction systems: Alignment-Based Learning vs. EMILE. Technical Report TR2001.05, University of Leeds, Leeds, UK, March 2001. 8

[238] Robert Wagner. Common phrases and minimum-space text storage. Communications of the ACM, 16(3), March 1973. 2.3.1

[239] JD Wang, Hsiang-Chuan Liu, Jeffrey JP Tsai, and Ka-Lok Ng. Scaling behavior of maximal repeat distributions in genomic sequences. International Journal of Cognitive Informatics and Natural Intelligence, 2(3):12, May 2008. 1.1.2

[240] Dennis Waters. The linguistic model in biology: Implications for recognizing life and intelligence. In Astrobiology Science Conference: Evolution and Life: Surviving Catastrophes and Extremes on Earth and Beyond, page 5368, January 2010. 1.1.1

[241] W Timothy J White and Michael D Hendy. Compressing DNA sequence databases with coil. BMC Bioinformatics, 9:242, January 2008. 2.3.3

[242] J Gerard Wolff. An algorithm for the segmentation of an artificial language analogue. British Journal of PsychologyJ, 66:79–90, 1975. 2.5, 2.6.7

[243] J Gerard Wolff. Learning syntax and meanings through optimization and distributional analysis. Categories and processes in language acquisition, January 1988. 1.2.1

[244] En-Hui Yang and John C Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. 1: Without context models. IEEE Transactions on Information Theory, 46 (3):755–777, May 2000. 2.3.4, 5, 5.3

[245] En-Hui Yang and John C Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. 2: With context models. IEEE Transactions on Information Theory, 49(11): 2874–2894, November 2003. 2.3.4, 5

[246] Daniel M Yellin. Algorithms for subset testing and finding maximal sets. In ACM-SIAM Symposium on Discrete Algorithms, pages 386–392, 1992. 5.4.1

[247] Sen Zhang, Ge Nong, and Wai Hong Chan. Fast and space efficient linear suffix array construction. In Data Compression Conference, Washington, DC, USA, 2008. IEEE Computer Society. 3.4.4

[248] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, January 1977. 2.6.1

[249] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory, 24(5): 530–536, January 1978. 2.6.2

VU :

Le Directeur de Thèse
(Nom et Prénom)

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)